Hashing & Hash Tables

CSE 332 Summer 2021

Instructor: Kristofer Wong

Teaching Assistants:

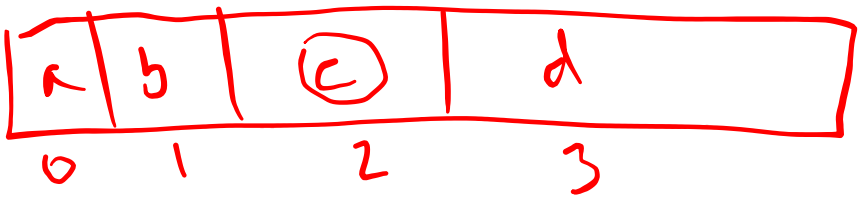
Alena Dickmann	Arya GJ	Finn Johnson
Joon Chong	Kimi Locke	Peyton Rapo
Rahul Misal	Winston Jodjana	

Announcements

- P2 Checkpoint 1; Gradescope issues
- Grading
  - Lecture activities
  - Ex 2 grades releasing this evening
- P2: uMessage
- UW on-campus guidance updates

gradescope gradescope.com/courses/275833

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

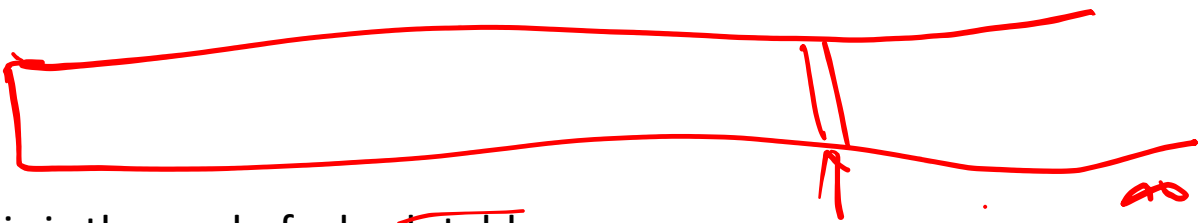


Lecture Outline

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

"Really Big Array"

- What if arrays didn't have size constraints?
- Easy mapping from index : value
- Runtime? O(1)



- This is the goal of a hash table

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 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();
}
```

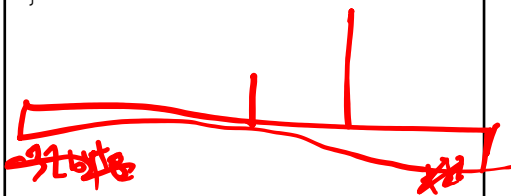
Deterministic? X

```
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? X

O(n²)

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



Uniform? X

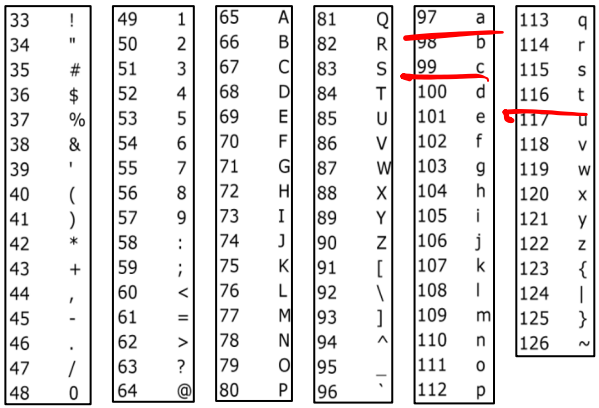
Attempt #1: hash("cat")

- 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

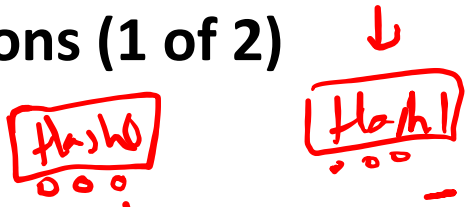


33	!	49	1	65	A	81	Q	97	a	113	q
34	"	50	2	66	B	82	R	98	b	114	r
35	#	51	3	67	C	83	S	99	c	115	s
36	$	52	4	68	D	84	T	100	d	116	t
37	%	53	5	69	E	85	U	101	e	117	u
38	&	54	6	70	F	86	V	102	f	118	v
39	'	55	7	71	G	87	W	103	g	119	w
40	(	56	8	72	H	88	X	104	h	120	x
41	)	57	9	73	I	89	Y	105	i	121	y
42	*	58	:	74	J	90	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		

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)



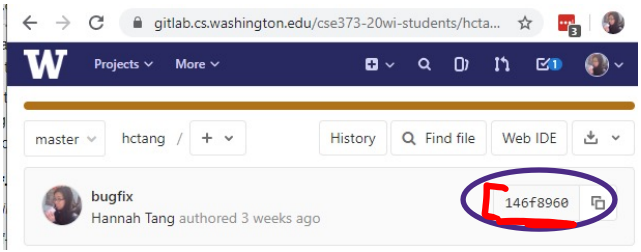
- Caching:
  - You've downloaded a large video file. You want to know if a new version is available. Rather than re-downloading the entire file, compare your file's hash value with the server's hash value.

- 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://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") ???

- 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**

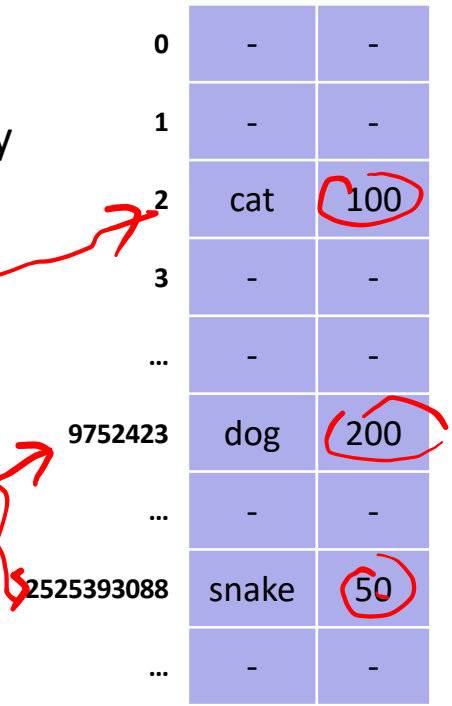
Hash Table: Idea (1 of 2)

- 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;
```



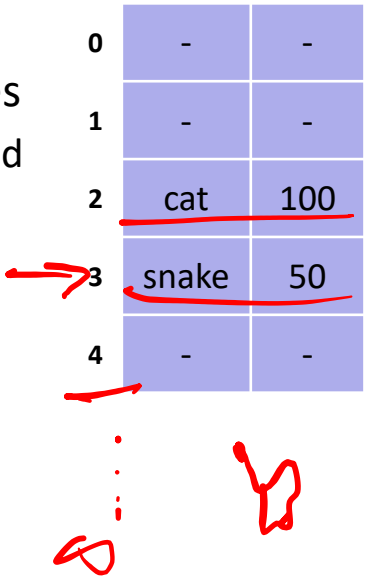
Hash Table: Idea (2 of 2)

- 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
```



gradescope

gradescope.com/courses/275833

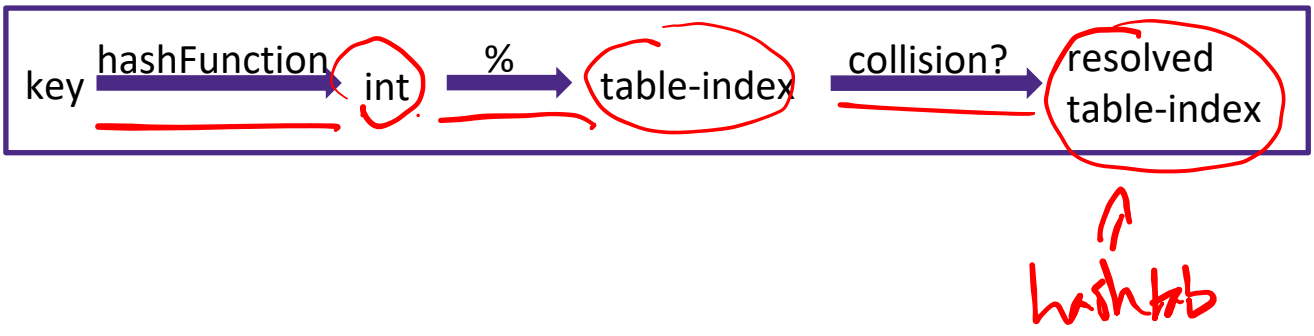
How should we handle the "snake" and "dog" collision at index 3?

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 $n << m$)
  - *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 % 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`

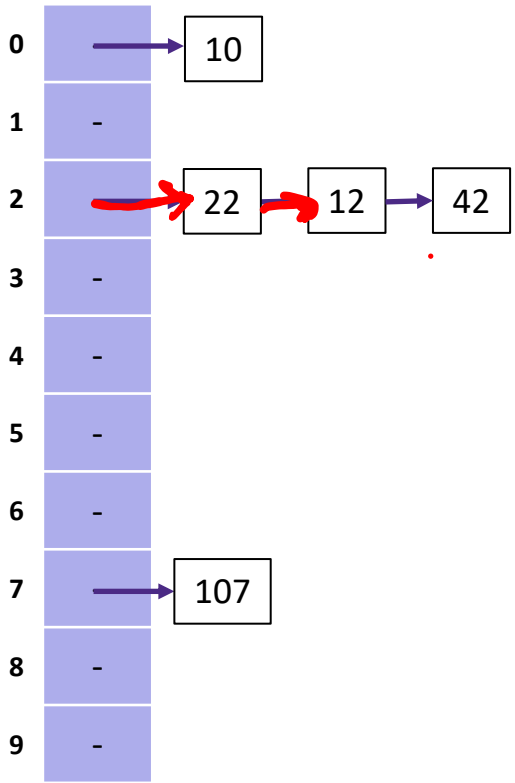
- Given table size `y` and key hashes as multiples of `x`, we'll get a decent distribution if `x` & `y` are co-prime
  - So choose a `TableSize` that has no common factors with any "likely pattern" `x`
  - And choose – don't implement – a decent hash function!

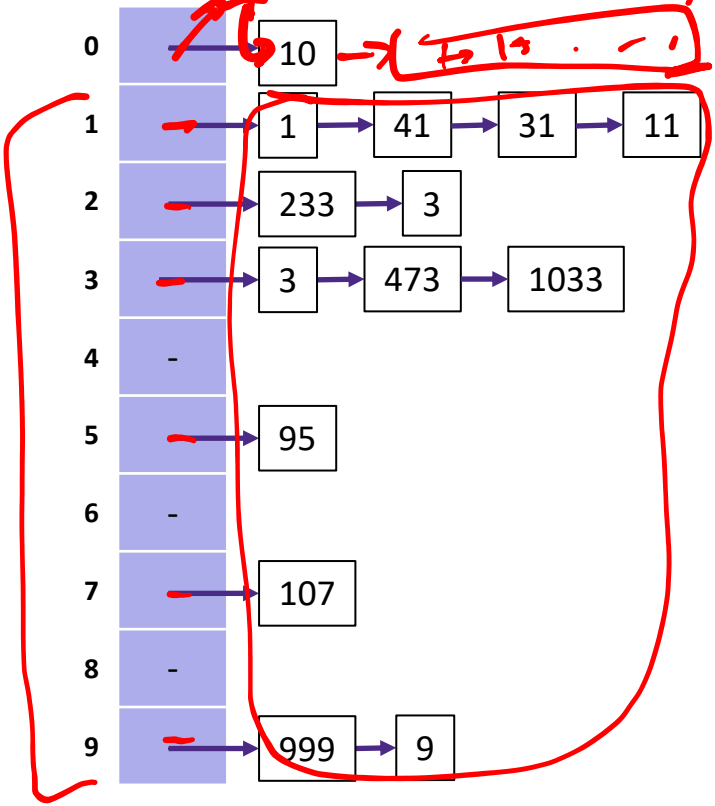
Lecture Outline

- Hashing != Hash Tables
  - Designing our own Hash Function
  - Hashing Applications
- Hash Tables
  - Introduction
  - Collision *Avoidance* Concepts
  - **Collision *Resolution*: Separate Chaining**
  - Collision *Resolution*: Open Addressing
  - Collision Avoidance: Rehashing
- Hash Tables & Java
- Conclusion

Separate Chaining and Load Factor

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





gradescope

gradescope.com/courses/275833

- What's the worst case runtime for ~~add~~ find & delete in separate chaining hash tables?

find O(n)

↳ O(n)

- Why do you think we care about **λ**?

add O(1)

Separate Chaining: Runtime

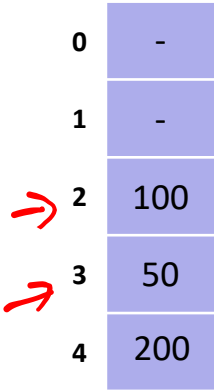
- It's not great..
- We typically use the load factor to determine when we should resize our table
- Depends on hash functions!!

Lecture Outline

- Hashing != Hash Tables
  - Designing our own Hash Function
  - Hashing Applications
- Hash Tables
  - Introduction
  - Collision *Avoidance* Concepts
  - Collision *Resolution*: Separate Chaining
  - **Collision *Resolution*: Open Addressing**
  - Collision Avoidance: Rehashing
- Hash Tables & Java
- Conclusion

Open Addressing Idea

- Why not use up the empty space in the table?
  - Store directly in the array cell (no linked list)
- How to deal with collisions?
  - If **`h(key) %TableSize`** is already full, …

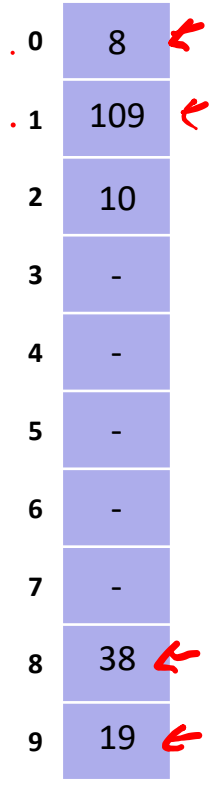



```
HashTable h;
h.add(100);
h.add(50);
h.add(200);
```

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

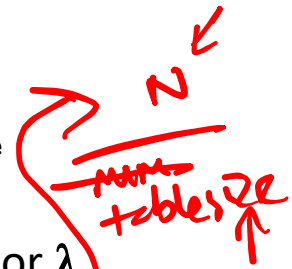
Linear Probing: Add Example

- Our first option for resolving this collision is *linear probing*
- If **`h(key)`** is already full,
  - try **`(h(key) + 1) % TableSize`**. If full,
  - try **`(h(key) + 2) % TableSize`**. If full,
  - try **`(h(key) + 3) % TableSize`**. If full…
- Example: add 38, 19, 8, 109, 10



Open Addressing

- **Open addressing** resolves collisions by trying a sequence of other positions in the table
  - Trying the *next* spot is called **probing**
  - We just did **linear probing**:
    - `i`th probe: `(h(key) + i) % TableSize`
  - In general have some **probe function** `f` and :
    - `i`th probe: `(h(key) + f(i)) % TableSize`
- Open addressing does poorly with high load factor λ
  - Typically want larger tables
  - Too many probes means no more $O(1)$ 😭😭😭

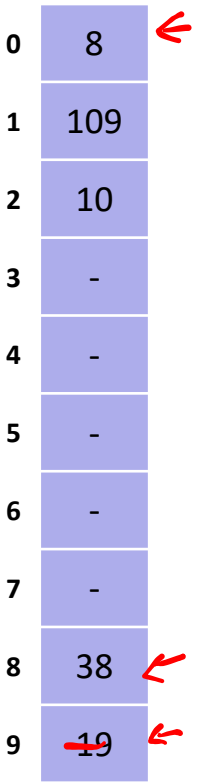


Linear Probing: find

- You can figure this one out too ☺
  - Must use same probe function to "retrace the trail" for the item
  - Unsuccessful search when reach empty position

- What is **`find`**'s runtime …
  - If key is NOT there? O(N)
  - Worst case?
  - If key is in table? O(N)

- What is **`find`**'s worst case runtime in a open addressing hash table:
  - If key is NOT there?
  - If the key is there?



Linear Probing: Remove

- remove(19)
- ***Must*** use "lazy deletion"
  - Marker/tombstone indicates "no item here, but don't stop probing"
  - Without lazy deletion, find() of an existing value is incorrect; with lazy deletion, find() runs in O(N)
- As with lazy deletion on other data structures, spots marked "deleted" can be filled in during subsequent adds

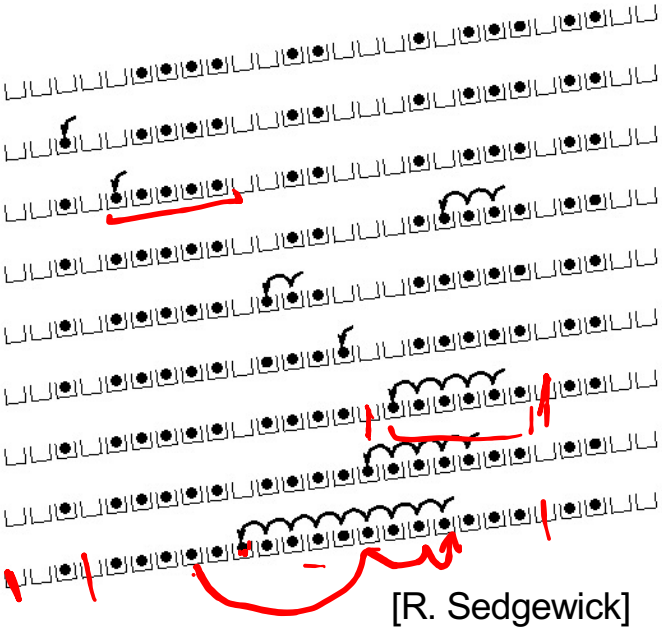
0	8	8
1	109	109
2	10	10
3	-	-
4	-	-
5	-	-
6	-	-
7	-	-
8	38	38
9	19	☠

Linear Probing: Primary Clustering

- It turns out linear probing is a *bad idea*, even though the probe function is quick to compute (a good thing)

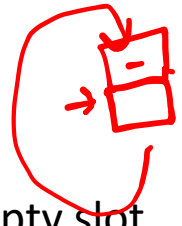
- Tends to produce *clusters*, which lead to long probe sequences
  - Called **primary clustering**

- Saw the start of a cluster in our linear probing example

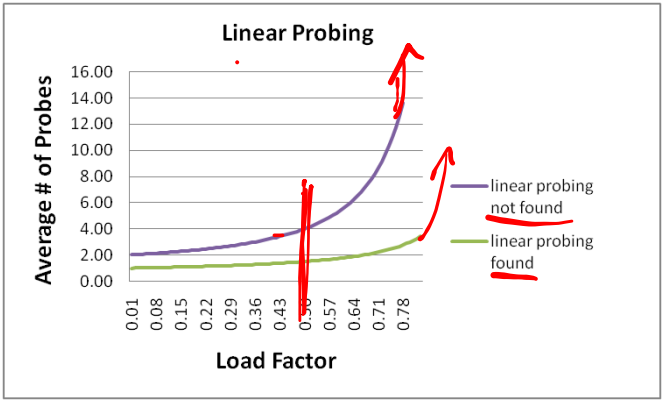


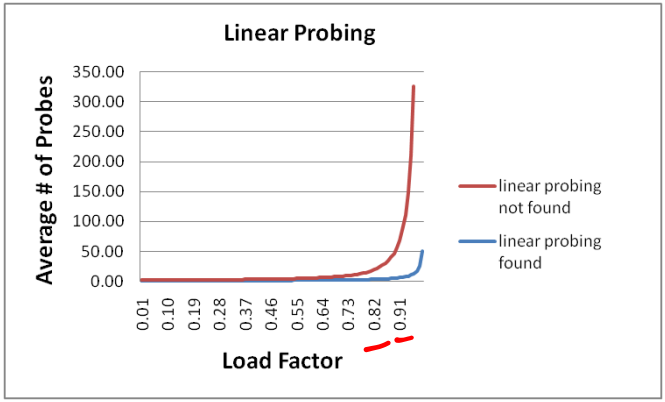
[R. Sedgewick]

Linear Probing: Analysis



- For any $\lambda < 1$, linear probing will find an empty slot
- Linear-probing performance degrades rapidly as table gets full
  - (Formula assumes "large table" but point remains)
  - With open addressing, a "good" λ to aim for is 0.5





- By comparison, separate chaining performance is linear in λ and has no trouble with $\lambda > 1$

Quadratic Probing

- Avoid primary clustering by changing the probe function:
  - i^{th} probe: `(h(key) + i^2) % TableSize`
  - Probe sequence becomes:
    - 0^{th} probe: `h(key) % TableSize`
    - 1^{st} probe: `(h(key) + 1) % TableSize`
    - 2^{nd} probe: `(h(key) + 4) % TableSize`
    - 3^{rd} probe: `(h(key) + 9) % TableSize`
- Intuition: Probes quickly "leave the neighborhood"

Quadratic Probing: Add Example

- Example: add 89, 18, 49, 58, 79
  - Let hashFunction(x) = x
  - Let `TableSize` = 10

0	49
1	-
2	58
3	79
4	-
5	-
6	-
7	-
8	18
9	89

Quadratic Probing: Another Add Example (1 of 3)

- Example: add 76, 40, 48, 5, 55, 47
  - Let hashFunction(x) = x
  - Let `TableSize` = 7

0	48
1	-
2	5
3	55
4	-
5	40
6	76

Quadratic Probing: Another Add Example (2 of 3)

- Example: add ~~76, 40, 48, 5, 55,~~ 47
  - Let hashFunction(x) = x
  - Let `TableSize` = 7

  - (47 + 1) % 7 = 6 collision!
  - (47 + 4) % 7 = 2 collision!
  - (47 + 9) % 7 = 0 collision!
  - (47 + 16) % 7 = 0 collision!
  - (47 + 25) % 7 = 2 collision!

  - **Will we ever get a 1 or 4?!?**

0	48
1	
2	5
3	55
4	
5	40
6	76

Quadratic Probing: Another Add Example (3 of 3)

- Example: add 76, 40, 48, 5, 55, 47

0	48
1	-
2	5
3	55
4	-
5	40
6	76

- Will we ever get a 1 or 4?!?
  - add(47) will *always* fail here. Why?
  - For all i, $(5 + i^2) \% 7$ is 0, 2, 5, or 6
  - Proof uses induction and
    - $(5 + i^2) \% 7 = (5 + (i - 7)^2) \% 7$
- In fact, for all c and k,
  - $(c + i^2) \% k = (c + (i - k)^2) \% k$

Quadratic Probing: Bad News / Good News

- Bad News:
  - After `TableSize` probes, we cycle through the same indices
- Good News:
  - If **`TableSize` is prime and $\lambda < ½$**, then quadratic probing will find an empty slot in at most **`TableSize/2`** probes
  - So: If you keep $\lambda < ½$ and **`TableSize`** is prime, no need to detect cycles
- Proof in following slides, won't go through it.

Quadratic Probing: Success Guarantee (1 of 2)

If `TableSize` is prime and $\lambda < ½$, then quadratic probing will find an empty bucket in `TableSize/2` probes or fewer

- Intuition: if the table is less than half full, then probing `TableSize/2` distinct buckets must find an empty one
  - Therefore, prove the first `TableSize/2` probes are distinct

Any i^{th} and any j^{th} probe results in a distinct bucket

- Theorem: for all **`0 ≤ i,j ≤ TableSize/2`**, and **`i ≠ j`**

`(h(x) + i²) % TableSize ≠ (h(x) + j²) % TableSize`

Quadratic Probing: Success Guarantee (2 of 2)

- Proof, by contradiction: suppose that for some $i \neq j$:

$$(h(x) + i^2) \% \text{TableSize} = (h(x) + j^2) \% \text{TableSize}$$
$$\Rightarrow i^2 \% \text{TableSize} = j^2 \% \text{TableSize}$$
$$\Rightarrow (i^2 - j^2) \% \text{TableSize} = 0$$
$$\Rightarrow [(i + j)(i - j)] \% \text{TableSize} = 0$$
$$\Rightarrow [(i + j)(i - j)] = k*\text{TableSize} \text{ for some } k \geq 1$$
or
$$[(i + j)(i - j)] = 0$$

CONTRADICTION!

- How can $i+j = 0$ or $i+j = k*\text{TableSize}$ when:
  $0 \leq i, j$ and $i \neq j$ and $i, j \leq \text{TableSize}/2$?
- How can $i-j = 0$ or $i-j = k*\text{TableSize}$ when
  $i \neq j$ and $i, j \leq \text{TableSize}/2$?

Quadratic Probing: Secondary Clustering

- Quadratic probing does not suffer from primary clustering!
  - We don't grow "big blobs" by adding to the end of a cluster

- Quadratic probing does not resolve collisions between different keys that hash *to the same* ***index***
  - These keys **have the same series of moves** looking for an empty spot
  - Called **secondary clustering** ☹

- Since the problem occurs when we have the different keys hashing to the same initial index, can we avoid secondary clustering with *a probe function that also incorporates the key*?
  - Known as **double hashing**

Double Hashing

- Double hashing:
  - `i`th probe: `(h(key) + i*g(key)) % TableSize`
  - Probe sequence becomes:
    - 0th probe: `h(key) % TableSize`
    - 1st probe: `(h(key) + g(key)) % TableSize`
    - 2nd probe: `(h(key) + 2*g(key)) % TableSize`
    - …
- Idea:
  - `g(key)` lets us "go different places from initial collisions"
    - It is very unlikely that for some key, `h(key) == g(key)`
    - (assuming good hash functions h and g)
  - `i*g(key)` lets us "leave the neighborhood"
- Detail: Ensure `g(key)` can't generate 0

Double Hashing: Add Example (1 of 3)

- Example: add 13, 28, 33, 147, 43
  - Remember: (`h(key) + i*g(key)) % TableSize`
  - Let h(x) = x%`TableSize`
  - Let g(x) = 1 + ((x/`TableSize`) % (`TableSize`-1))
  - Let `TableSize` = 10

0	-
1	-
2	-
3	13
4	33
5	-
6	-
7	147
8	28
9	-

Double Hashing: Add Example (2 or 3)

- Example: add ~~13, 28, 33, 147,~~ 43
  - Remember: `(h(key) + i*g(key)) % TableSize`
  - Let h(x) = x%`TableSize`
  - Let g(x) = 1 + ((x/`TableSize`) % (`TableSize`-1))
  - Let `TableSize` = 10

  - h(43) = 3 and g(43) = 1 + (4%9) = 5
  - 3 + 0*5 = 3 collision!
  - 3 + 1*5 = 8 collision!
  - 3 + 2*5 = 13 collision
  - **Will we ever get anything else?!?**

0	-
1	-
2	-
3	13
4	33
5	-
6	-
7	147
8	28
9	-

Double Hashing: Add Example (3 of 3)

- Example: add ~~13, 28, 33, 147,~~ 43
  - Remember: (`h(key) + i*g(key)) % TableSize`
  - Let h(x) = x%`TableSize`
  - Let g(x) = 1 + ((x/`TableSize`) % (`TableSize`-1))
  - Let `TableSize` = 10
  - Will we ever get anything else?!?
    - No. add(43) will always fail here. Why?

0	-
1	-
2	-
3	13
4	33
5	-
6	-
7	147
8	28
9	-

Double Hashing: Considerations (1 of 2)

- Our example implies the possibility of infinite probe sequences ☹
  - But we can be avoid infinite probes if our functions are:
    - h(key) = hash1(key) % p
    - g(key) = q – (hash2(key) % q)
    - p and q are primes, with 2 < q < p

Double Hashing: Considerations (2 of 2)

- Double hashing:
  - **`i`**th probe: **`(h(key) + i*g(key)) % TableSize`**

- Assume **`g(key)`** divides `TableSize`
  - That is, there exists some integer `x` such that `x*g(key)=TableSize`
  - Therefore: after x probes, we'll "loop through" the same indices as before
  - Example:
    - `TableSize=50`
    - `g(key)=25`
    - Probe sequence:
      - i=0: h(key)
      - i=1: h(key)+25
      - i=2: h(key)+50 = h(key)
      - i=3: h(key)+75 = h(key)+25
      - …

- Bottom line: don't let **`g(key)`** divide `TableSize`
  - That is, choose a prime `TableSize` when using double hashing

Double Hashing: Performance

- Assume g() distributes its keys uniformly over its range
  - That is: probability of `g(key1)%p == g(key2)%p` is `1/p`
- We won't prove the following:
  - Average # of probes (in the limit as TableSize →∞), **unsuccessful** find:

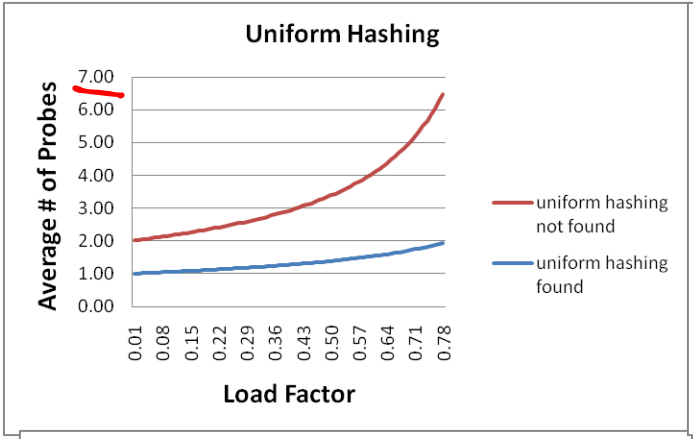
$$\frac{1}{1-\lambda}$$

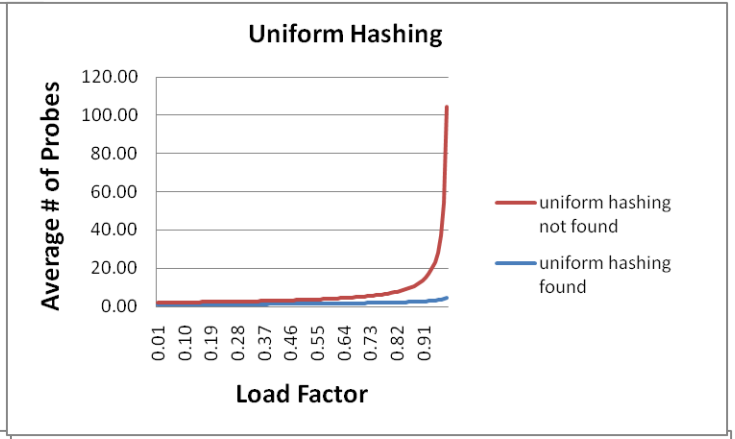
  - Average # of probes (in the limit as TableSize →∞), **successful** find:

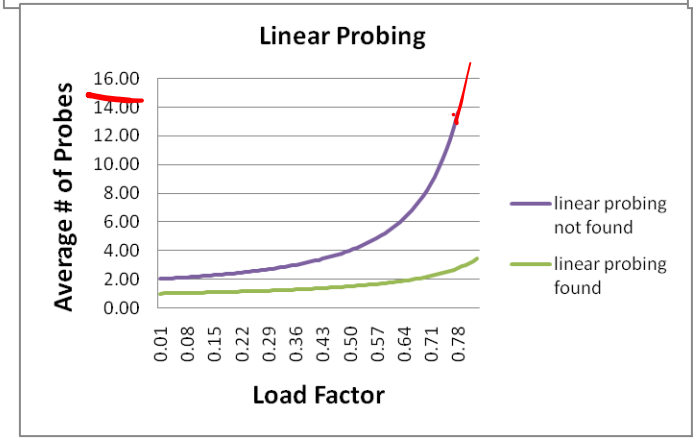
$$\frac{1}{\lambda}\log_e\left(\frac{1}{1-\lambda}\right)$$

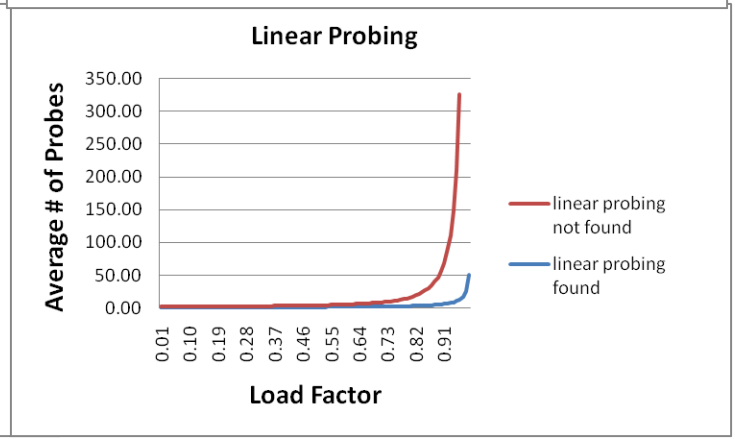
- Bottom line:
  - Performance of unsuccessful finds degrades with λ (but not as quickly as linear probing degrades)
  - Performance of successful finds degrades not nearly as quickly

Double Hashing vs Linear Probing Performance









Lecture Outline

- Hashing != Hash Tables
  - Designing our own Hash Function
  - Hashing Applications
- Hash Tables
  - Introduction
  - Collision *Avoidance* Concepts
  - Collision *Resolution*: Separate Chaining
  - Collision *Resolution*: Open Addressing
  - **Collision Avoidance: Rehashing**
- Hash Tables & Java
- Conclusion

Separate Chaining vs Open Addressing

- Separate Chaining
  - **`find, add, remove`** proportional to λ if using unsorted LL
  - If using another data structure for buckets (e.g. AVL tree) , runtime is proportional to runtime for that structure

- Open addressing: has clustering issues as table fills ($\lambda > 1/2$)
  - Why use it:
    - Some runtime for allocating nodes; open addressing could be faster?
    - Easier data representation?

Rehashing (1 of 3)

- As with array-based stacks/queues/lists, if table gets too full, create a bigger table and "copy" everything over
- With separate chaining, we decide what "too full" means
  - Keep load factor reasonable (e.g., < 2)?
  - Consider average or max size of non-empty chains?
- For open addressing, half-full is a good rule of thumb

Rehashing (2 of 3)

- Can't actually copy to the same indices in the new table
  - We'd calculated the index based on **`TableSize`**
- For each key/value in old table, must `add` into new table
  - Iterate over old table: O(n)
  - n calls to the hash function: $n \cdot O(1) = O(n)$
- Can we avoid all those hash function calls?
  - Space/time tradeoff: Could store `h(key)` with each item
  - Iterating over the table is still O(n); saving `h(key)` only helps by a constant factor

Rehashing (3 of 3)

- New table size
  - Twice-as-big is a good idea, except … ummm … that won't be prime!
  - So go *about* twice-as-big
    - Hard-coded list of primes (you probably won't grow more than 20-30 times)
    - Calculate primes after that

11
23
47
.
.
.

Lecture Outline

- Hashing != Hash Tables
  - Designing our own Hash Function
  - Hashing Applications
- Hash Tables
  - Introduction
  - Collision *Avoidance* Concepts
  - Collision *Resolution*: Separate Chaining
  - Collision *Resolution*: Open Addressing
  - Collision Avoidance: Rehashing
- **Hash Tables & Java**
- Conclusion

Hashing and Equality Testing

- Our examples use an `int` key, which overlooks a critical detail:
  - We hash **`K`** to get a table index
  - While chaining or probing, we need to test whether the current **`K'`** is equal to the **`K`** we're looking for
- So a Java hash table needs a hash *and* an equality function
  - Fortunately, in Java every object defines an **`equals`** and a **`hashCode`** method

```
class Object {
  boolean equals(Object o) {…}
  int hashCode() {…}
  …
}
```

Overriding equals()? Override hashCode() too

- The Java library (and your project's hash table) make a very important assumption that *all* clients must satisfy:

  If `a.equals(b)`, then `a.hashCode() == b.hashCode()`

- In other words, if you ever override equals:
  - You must also override hashCode() in a consistent way
  - See Core Java book, Ch. 5, for other "gotchas" with equals()

compareTo() rules

- Java also makes assumptions about `compareTo()` that affect:
  - All our dictionaries
  - Sorting (next major topic)

- Comparison must impose a consistent, total ordering:
  - For all **`a`**, **`b`**, and **`c`**,
    - If **`a.compareTo(b) < 0`**, then **`b.compareTo(a) > 0`**
    - If **`a.compareTo(b) == 0`**, then **`b.compareTo(a) == 0`**
    - If **`a.compareTo(b) < 0`** and **`b.compareTo(c) < 0`**, then **`a.compareTo(c) < 0`**

A Generally-Good hashCode()

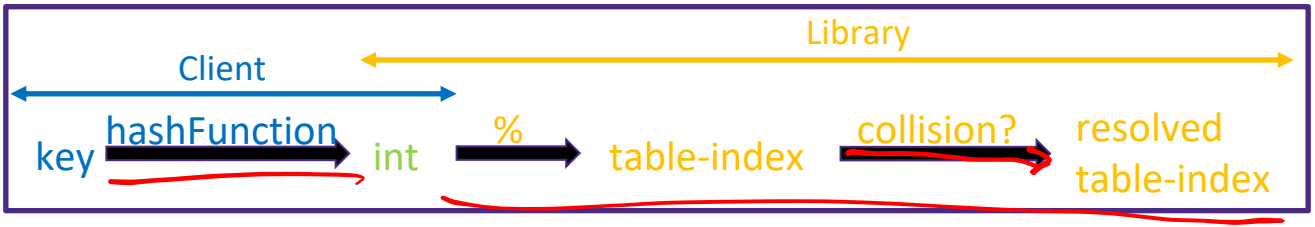
```
int result = 17;  // start at a prime

foreach field f
   int fieldHashcode =
     boolean: (f ? 1: 0)
     byte, char, short, int: (int) f
     long: (int) (f ^ (f >>> 32))
     float: Float.floatToIntBits(f)
     double: Double.doubleToLongBits(f),
         then above conversion to int
     Object: object.hashCode()
   result = 31 * result + fieldHashcode;

return result;
```

Lecture Outline

- Hashing != Hash Tables
  - Designing our own Hash Function
  - Hashing Applications
- Hash Tables
  - Introduction
  - Collision *Avoidance* Concepts
  - Collision *Resolution*: Separate Chaining
  - Collision *Resolution*: Open Addressing
  - Collision Avoidance: Rehashing
- Hash Tables & Java
- **Conclusion**

Who Hashes What?

- When used as a library, hash tables generally have two roles: client vs library



- We learned both, but you'll spend more time as clients
  - Both roles must contribute to minimizing collisions
  - Client should aim for different ints for expected keys
    - Avoid "wasting" any part of K or the int's bits
  - Library should aim for putting "similar" ints in different indices
    - Conversion to index is almost always "mod table-size"
    - Using prime numbers for table-size is common

Summary: Hash Tables vs. Balanced Trees

- In terms of a Dictionary ADT for just **`add`**, **`find`**, **`remove`**, hash tables and balanced trees are just different data structures
  - Hash tables $O(1)$ on average (assuming few collisions)
  - Balanced trees $O(\log n)$ worst-case

- Constant-time is better, right?
  - Yes, but you need "hashing to behave" (must avoid collisions)
  - Yes, but what if we want to **`findMin`**, **`findMax`**, **`predecessor`**, and **`successor`**, **`printSorted?`**
    - Hash tables are not designed to efficiently implement these operations
    - Your textbook considers hash tables to be a different ADT; not so important to argue over the definitions

Summary: Hash Table (1 of 2)

- Hash tables are categorized by collision *resolution* strategy:
  - **Separate chaining**: use an auxiliary data structure so that colliding keys can both use the same index
    - Simple is best (eg, linked list, or LL + an extra key/value slot)
    - λ can be > 1, but recommend keeping it "smallish"
  - **Open addressing**: look elsewhere in the array if keys collide. $\lambda \leq 1$
    - **Linear probing**: finds a slot if $\lambda < 1$, but primary clustering severely impacts performance (secondary clustering is also a consideration)
    - **Quadratic probing**: finds a slot if $\lambda < 0.5$. No primary clustering but secondary clustering is possible
    - **Double hashing**: finds depending on how h(x) and g(x) are constructed.

Summary: Hash Table (2 of 2)

- Collision *avoidance* applicable to both types of hash table
  - **Crucial** to use a good hash function: deterministic, fast, uniform
  - Which fields to hash is **important**: need "just enough" differentiation
  - Array size is **important**:
    - Choose a prime size
    - "Preferred λ" depends on type of table; resize (rehash) to maintain

- The hash table is one of the most important data structures
  - Useful in many, many, many real-world applications