

COMP\_SCI 214: Data Structures and Algorithms

# Hash Tables

PROF. SRUTI BHAGAVATULA

# Announcements

- ▶ Worksheets due today
- ▶ Homework 2 self-eval due today
- ▶ Homework 2 resubmission due today
- ▶ Homework 3 to be released later tonight
  - ▶ Due Tuesday, Feb 6<sup>th</sup> so you have enough time to prepare for the exam AND HW3!
  - ▶ Deadlines shift to Tuesdays starting with HW3 (except for finals week)

# Exam 1

- ▶ **Next Thursday, Feb 1<sup>st</sup> during class time in this room**
- ▶ Topics: Lecture 1 upto and including Lecture 7 (today)
- ▶ Exam 1 practice exam to be released today or tomorrow
  - ▶ Do not overfit your studying to this exam!

# Heads up for next Tuesday

- ▶ We will begin graphs on Tuesday
  - ▶ Assuming you have read Chapter 10 of the textbook and completed your worksheets
  - ▶ If you don't do this, you'll be lost as we'll pick up from there
- ▶ We'll start next class with some time for Q&A

# Motivating scenario

- ▶ I have an address book with all my friends' information (address, phone number, birthday, etc.). I'd like to be able to store all my friends' information and also lookup a friend's information easily.
- ▶ How can I do this?



# ADT: Dictionary

► Abstract values look like:

► {'Aparna': 'SF', 'Maria': 'Chicago', 'Aditi': 'Bengaluru'}

► Dictionary interface:

```
interface DICT[K, V]: # key and value types up to client
  def mem?(self, key: K) -> bool?
  def get(self, key: K) -> V
  def put(self, key: K, value: V) -> NoneC
  def del(self, key: K) -> NoneC
```

# Dictionary implementations so far

	Unsorted array of (key, value) pairs	Sorted array of (key, value) pairs; sorted by key	Linked list of (key, value) pairs
<b>Lookup</b>	$O(n)$	$O(\log n)$	$O(n)$
<b>Insert</b>	$O(n)$	$O(n)$	$O(n)$
<b>Delete</b>	$O(n)$	$O(n)$	$O(n)$
<b>Member?</b>	$O(n)$	$O(\log n)$	$O(n)$

# Dictionary implementations so far

Can we do better?

L			
<b>Insert</b>	$O(n)$ $O(1)$ amortized	$O(n)$	$O(n)$
<b>Delete</b>	$O(n)$	$O(n)$	$O(n)$
<b>Member?</b>	$O(n)$	$O(\log n)$	$O(n)$



# Revisiting direct addressing

# Recall: Array as a simple dictionary

- ▶ An array already implements a very basic dictionary
  - ▶ Key: array index (type: always integer  $\geq 0$ )
  - ▶ Value: element value at index (type: whatever type the array holds)

	0	1	2	3	4	5
arr	'sr'	'ut'	'i'	'oi'	'n'	'k'

- ▶ Called "Direct Addressing" when the index is the key

# Direct addressing

- ▶ Key as index directly is convenient to get  $O(1)$  lookups and updates
- ▶ But our keys are more complex than simple integers
- ▶ Could we try to get the best of both worlds?
  - ▶ We tried with sorted and unsorted arrays but they were kind of slow
- ▶ What if we could find a way to convert a complex key to an integer index?
  - ▶ And then do direct addressing

# Hash tables

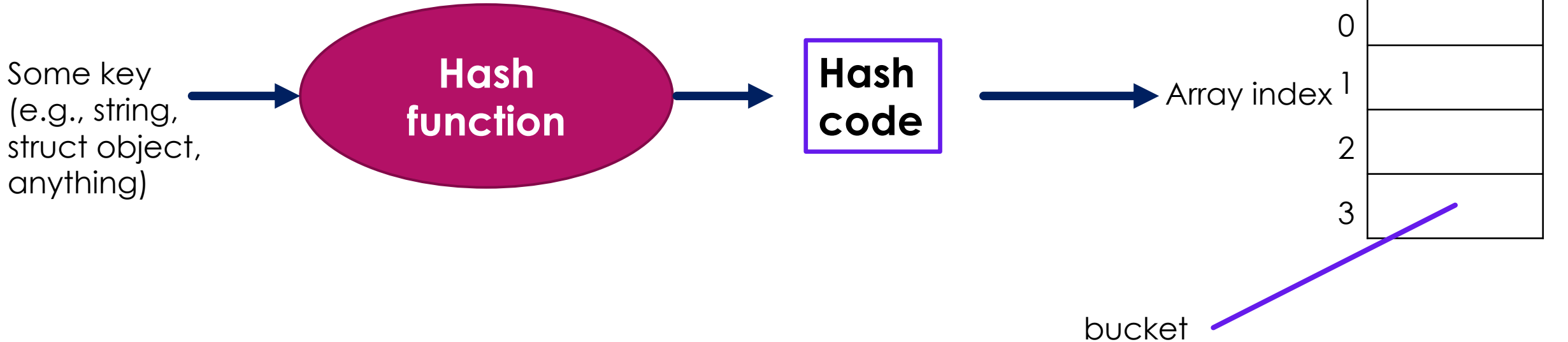


# Vocabulary to start with

- ▶ **Hash function**: A function that maps a key (any type) to an integer
- ▶ **Hash code**: The result of a hash function
  - ▶ This could become the index into an array
- ▶ **Hash table**: An array where entries can be stored at the hash code index
  - ▶ A concrete data structure
  - ▶ We'll call each element of this array a **bucket**



# A hash table to implement a dictionary



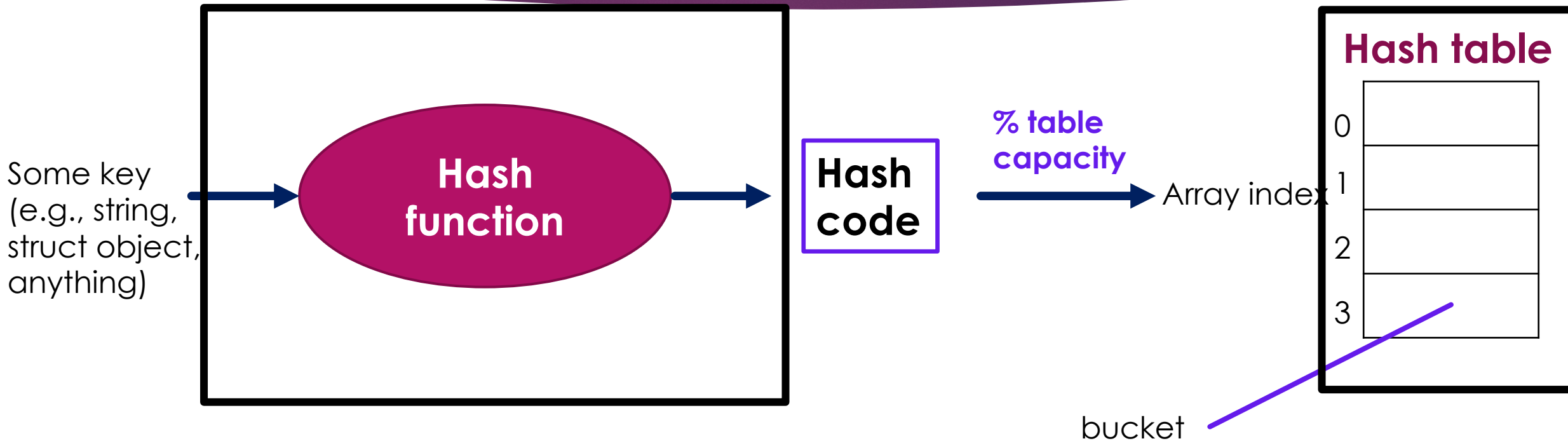
# Hash function outputs

- ▶ What if your hash code is an integer that doesn't map to any of your array indices?
  - ▶ E.g., if your array capacity (number of elements) is 6 and your hash function outputs the code 4568
- ▶  $\text{Array index} = \text{hash code} \% \text{capacity}$ 
  - ▶ For above example, array index = 2

# Updated vocabulary

- ▶ **Hash function**: A function that maps a key (any type) to an integer
- ▶ **Hash code**: The result of a hash function
  - ▶ This can be converted into a bucket index
- ▶ **Hash table**: An array where entries can be stored at the bucket index
  - ▶ A concrete data structure
  - ▶ Each element of this array is called a **bucket**

# A hash table to implement a dictionary



- Two questions we need to address:
1. What can the hash function look like?
  2. How do we store entries in a bucket?

# Example: Address book

{'Aparna': 'SF', 'Maria': 'Chicago', 'Aditi': 'Bengaluru'}

► Key is a string holding a friend's name

1. How can we hash names to get an integer?

► **Hashing attempt #1:** Use the position in the alphabet of the first letter of the first person's name as our integers

► A = 0, B = 1, C = 2, ...

2. How can we store an entry in the hash table?


► Each bucket holds a single key-value pair



# The “first letter” hash table

- ▶ To look up a key:
  - ▶ Hash the key, look in the correct bucket, and get the value
- ▶ To insert a key:
  - ▶ Hash the key, add key-value pair to the bucket

One key-value pair entry




<b>(bucket)</b>	<b>Name</b>	<b>City, other info, etc.</b>
(0)	Aparna	San Francisco
(1)	Branden	Evanston
(2)	Connor	Evanston
...	...	...
(12)	Maria	Chicago
...	...	...
(15)	Pardis	Durham

# The “first letter” hash table

- ▶ What if I want to add two new friends:
  - ▶ Aditi
  - ▶ Mahmood
- ▶ Where would they go in the table?
  - ▶ Those spots are filled
  - ▶ Bucket can only hold one entry

One key-value pair entry



<b>(bucket)</b>	<b>Name</b>	<b>City, other info, etc.</b>
(0)	Aparna	San Francisco
(1)	Branden	Evanston
(2)	Connor	Evanston
...	...	...
(12)	Maria	Chicago
...	...	...
(15)	Pardis	Durham

# Hash collision!

- ▶ Using our "first letter" hash function  $h_1$

$$h_1(\text{"Aparna"}) = 0$$

$$h_1(\text{"Maria"}) = 12$$

$$h_1(\text{"Branden"}) = 1$$

$$h_1(\text{"Aditi"}) = \text{also } 0!$$

- ▶ When a hash function gives the same value for two keys, a **hash collision** occurs

# How can we resolve these collisions?

- ▶ Our hash function  $h_1$  is really bad so just use a better hash function?
  - ▶ Even with much better hash functions, collisions are inevitable
- ▶ Need resolutions for the collisions when they occur

# How can we resolve these collisions?

- ▶ Store a linked list in each bucket (separate chaining)
  - ▶ Your Homework 3
- ▶ Use some other free bucket instead (open addressing)
  - ▶ Linear probing, quadratic probing, double hashing
- ▶ Plus variants on top of these options!
- ▶ Lots of approaches with different tradeoffs!
  - ▶ All same concrete data structure of hash tables!



# We'll see two today

- ▶ Store a linked list in each bucket (**separate chaining**)
  - ▶ Your Homework 3
- ▶ Use some other free bucket instead (**open addressing**)
  - ▶ Linear probing, quadratic probing, double hashing
- ▶ Plus variants on top of these options
- ▶ Lots of approaches with different tradeoffs!
  - ▶ All same concrete data structure of hash tables!

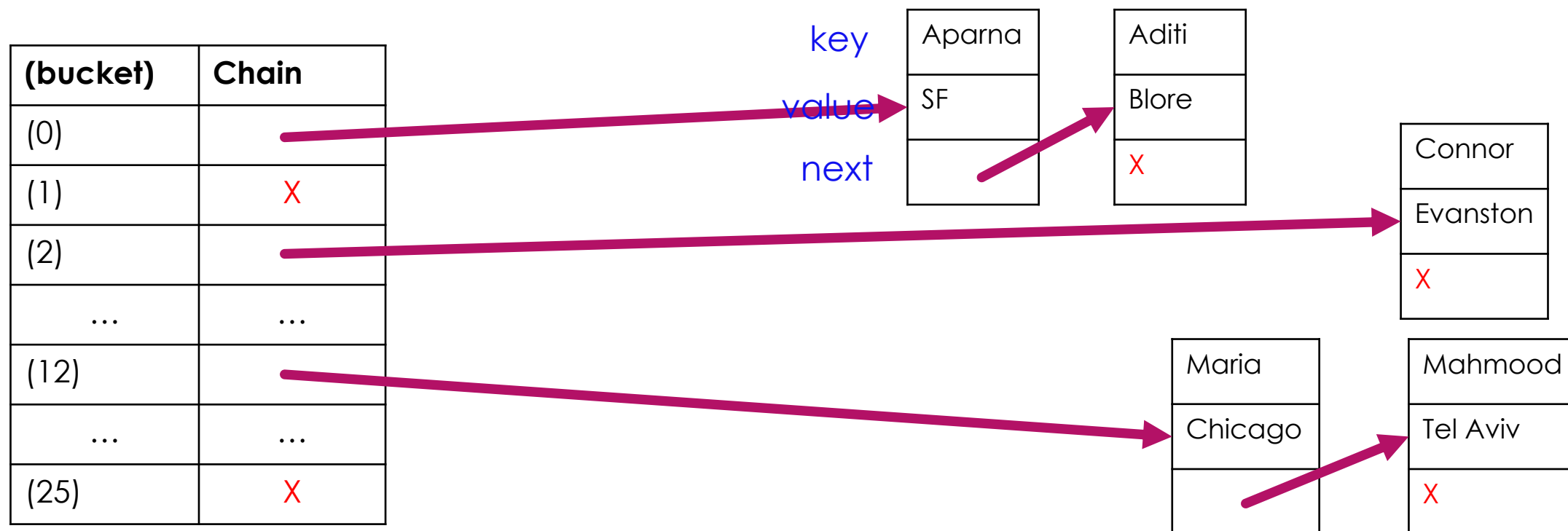
# Pause

- ▶ Any questions?
- ▶ Anything unclear or that I missed?

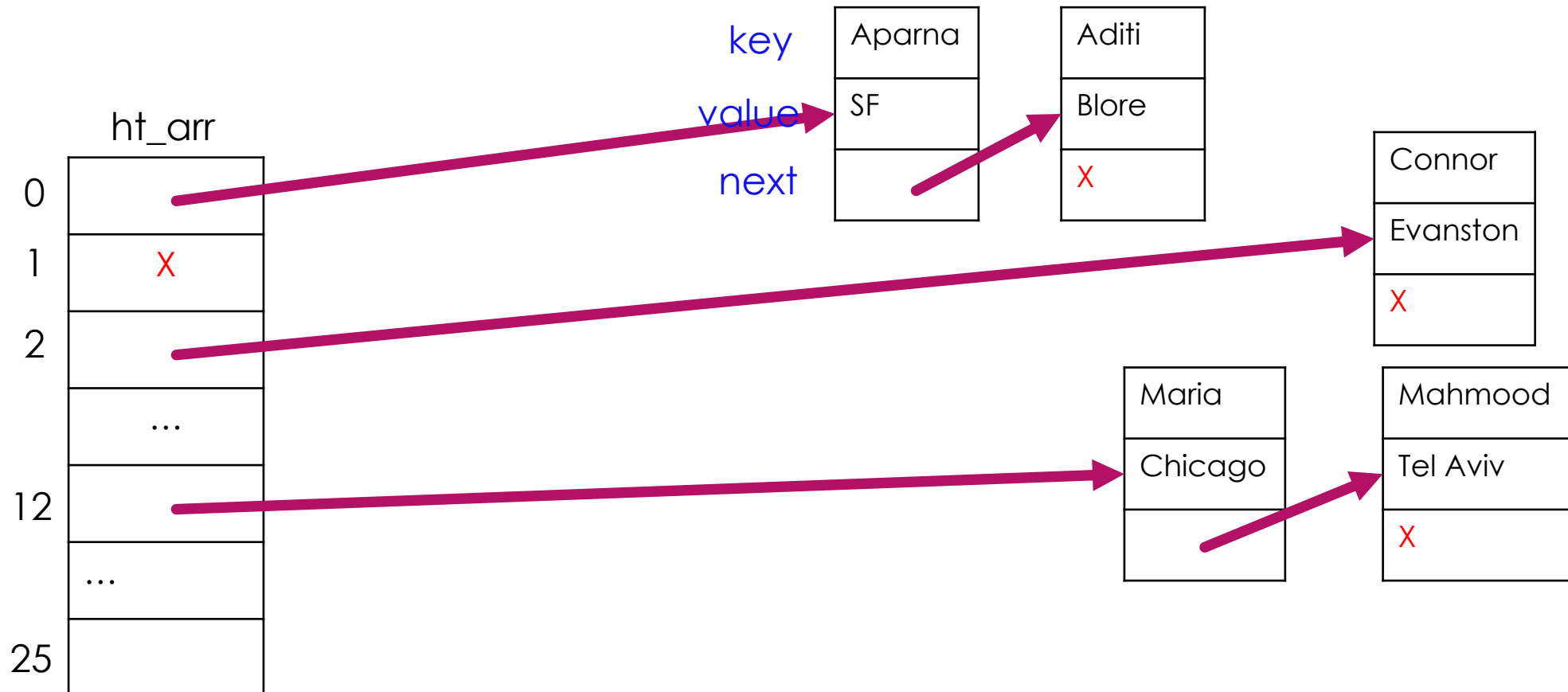
# Collision resolution

# Separate chaining

- One bucket stores a linked list of key-value pairs that hash to same bucket



# How the actual hash table array would look





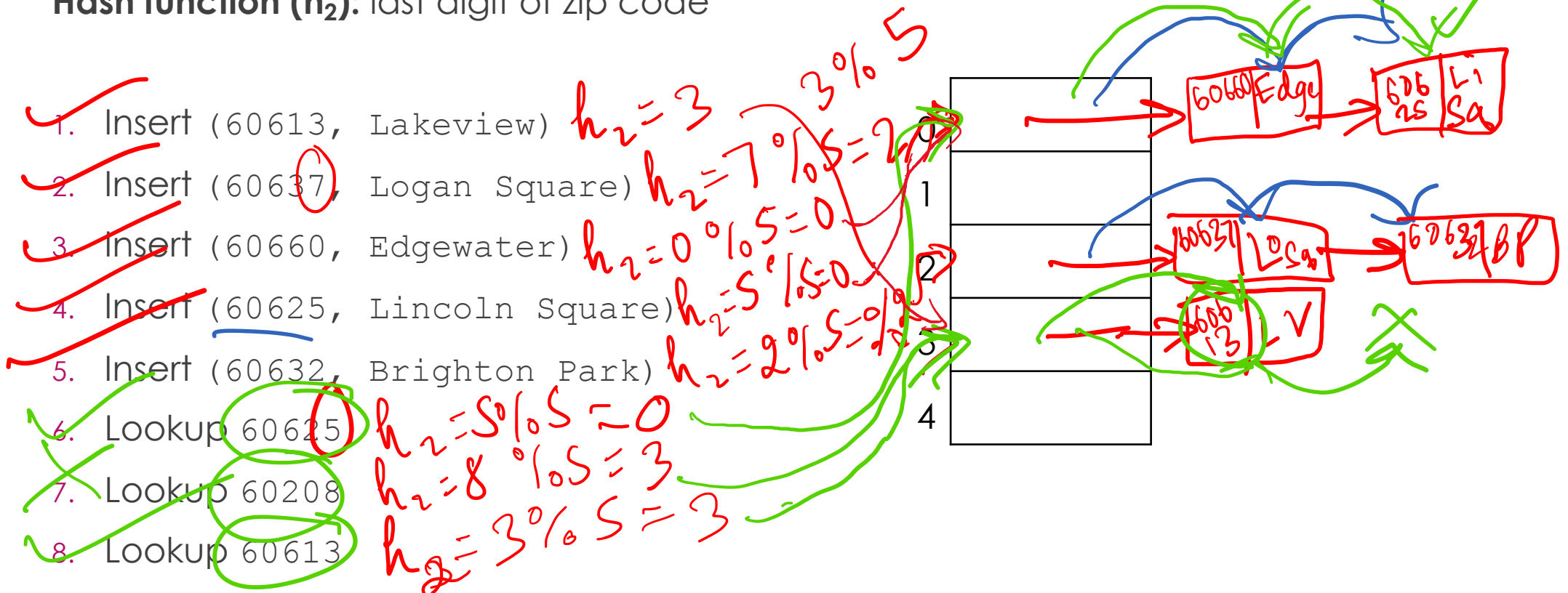
# Separate chaining

- ▶ How do we insert a key?
  1. Find the right bucket
  2. Traverse the list in that bucket to make sure the key doesn't exist
    - ▶ Remember: different keys can hash to same bucket!
  3. Insert key-value pair at the end or beginning of list
- ▶ Insert, delete, and membership-check work in a similar way
- ▶ Does anything seem familiar? Hash table is like an array of association lists!

# Practice with operations

**Dictionary:** mapping zip codes to Chicago neighborhoods

**Hash function ( $h_2$ ):** last digit of zip code



# Open addressing (Linear probing)

- ▶ After hashing, if the bucket is already occupied, look at another bucket

0	
1	
2	
3	

# Open addressing: Linear probing

- ▶ How do we insert a key?
  - ▶ Start at "correct" bucket, look at the next bucket until we find an empty bucket (circle around if needed); insert entry in that bucket
- ▶ When looking up:
  - ▶ Start at "correct" bucket, keep looking at the next buckets until you find the key you're looking for
    - ▶ If you reach an empty bucket or original bucket, key is not in hash table
- ▶ Delete and membership-check are very similar

# Practice with operations

**Dictionary:** mapping zip codes to Chicago neighborhoods

**Hash function ( $h_2$ ):** last digit of zip code

Capacity = 4

0	60637, Lo Sq
1	60660, EW
2	60625, Li Sq
3	60613, Lakeview

Handwritten operations and calculations:

- Insert (60613, Lakeview)  $h_2 = 3 \% 4 = 3$
- Insert (60637, Logan Square)  $h_2 = 7 \% 4 = 3$
- Insert (60660, Edgewater)  $h_2 = 0 \% 4 = 0$
- Insert (60625, Lincoln Square)  $h_2 = 5 \% 4 = 1$
- Insert (60632, Brighton Park)  $h_2 = 2 \% 4 = 2$
- Lookup 60625  $h_2 = 5 \% 4 = 1$
- Lookup 60208  $h_2 = 8 \% 4 = 0$
- Lookup 60613

Diagram illustrating the hash function  $h_2$  (last digit of zip code) mapping to the array indices (0 to 3) and the corresponding neighborhoods stored in the array. The array has a capacity of 4. The neighborhoods are: 60637, Lo Sq (index 0); 60660, EW (index 1); 60625, Li Sq (index 2); 60613, Lakeview (index 3). Arrows show the mapping from the last digit of the zip code to the index and then to the neighborhood name.



# Representation invariants for separate-chaining hash table

- ▶ Entries are placed at the correct index by the hash-modulo operation
- ▶ Chains are separate
- ▶ Chains are acyclic
- ▶ ...
- ▶ <Add more based on your specific implementation>

# Hash table example code

- Implementation for linear probing is in `hash.rkt` on Canvas

# Pause

- ▶ Any questions?
- ▶ Anything unclear or that I missed?

# Check your understanding

- ▶ Is the hash code always the same as the bucket index?
  - ▶ No:  $(\text{hash\_code} \bmod \text{capacity})$  is the bucket index
- ▶ By resolving hash collisions, are we allowing duplicate keys?
  - ▶ No: Distinct keys can hash to the same code and thus result in a collision

# Complexity of hash tables



# Operations

- ▶ Lookup
  - ▶ Insert
  - ▶ Delete
  - ▶ Member?
- 
- ▶ Every operation is at least as expensive as lookup

# Complexity analysis

- ▶ Setup:
  - ▶ Table contains  $n$  entries
  - ▶ Table has capacity  $m$  (# of buckets)
  
- ▶ Two versions of hash table to analyze
  1. Separate chaining table
  2. Linear probing table

# Complexity analysis

- ▶ Setup:
  - ▶ Table contains  $n$  entries
  - ▶ Table has capacity  $m$
  
- ▶ Two versions of hash table to analyze
  1. Separate chaining table
  2. Linear probing table

# In-class exercise (5 minutes)

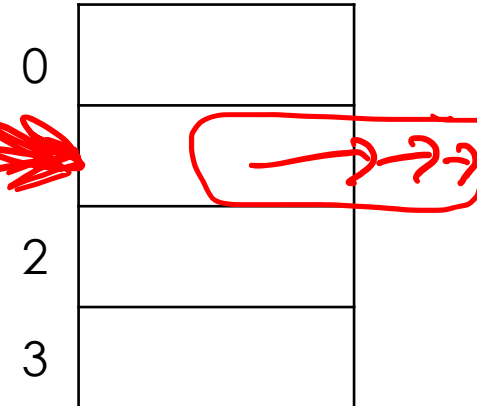
1. Given a hash table of capacity ( $m=4$ ) and the identity function as the hash function ( $f(x) = x$ ), which of the following operations would result in a collision if executed sequentially? List the letters.

a) Insert key 37

b) Insert key 41

c) Insert key 101

d) Insert key 89



2. If you resolve collisions using separate chaining, is the worst-case complexity of lookup in this hash table better, worse, or the same as an association list (linked list of key-value pairs)? Explain why.

# Worst possible layout

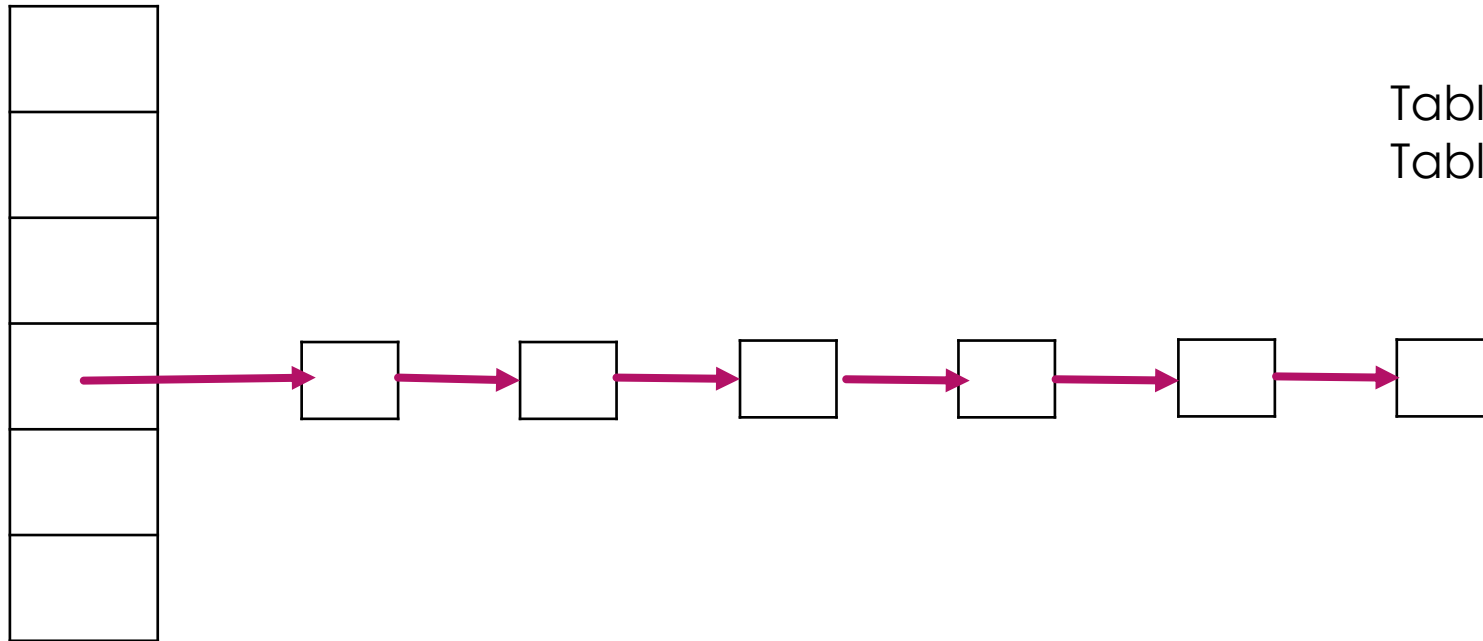


Table contains  $n$  entries  
Table has capacity  $m$

- Worst-case lookup complexity:  $O(n)$



# Best possible layout

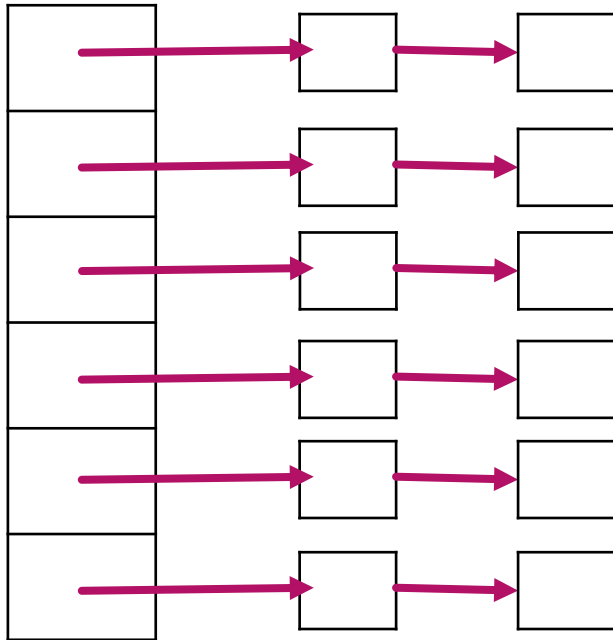


Table contains  $n$  entries  
Table has capacity  $m$

- Best-case lookup complexity:  $O(\frac{n}{m})$

# Complexity analysis

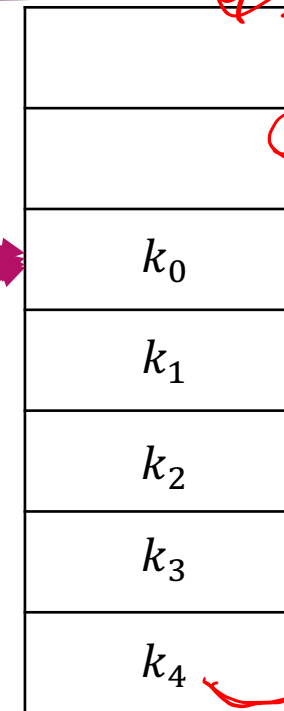
- ▶ Setup:
  - ▶ Table contains  $n$  entries
  - ▶ Table has capacity  $m$
- ▶ Two versions of hash table to analyze
  1. Separate chaining table
  2. Linear probing table

# Worst possible layout

- ▶  $h(k_0) \% m \Rightarrow idx_0$
- ▶  $h(k_1) \% m \Rightarrow idx_0$
- ▶  $h(k_2) \% m \Rightarrow idx_0$
- ▶  $h(k_3) \% m \Rightarrow idx_0$
- ▶  $h(k_4) \% m \Rightarrow idx_0$

## ▶ Where:

- ▶  $k_i$  and  $idx_i$  are arbitrary
- ▶  $k_0 \neq k_1 \neq k_2 \neq k_3 \neq k_4$



$k_0$
$k_1$
$k_2$
$k_3$
$k_4$

Table contains  $n$  entries  
Table has capacity  $m$

$$n \leq m$$

Looking up  $k_4$ :

- Go to  $idx_0$  (not there)
- Look at  $n$  elements before finding  $k_4$

➤ Worst-case lookup complexity:  $O(n)$

# Best possible layout

►  $h(k_0) \% m \Rightarrow idx_0$

►  $h(k_1) \% m \Rightarrow idx_1$

►  $h(k_2) \% m \Rightarrow idx_2$

►  $h(k_3) \% m \Rightarrow idx_3$

►  $h(k_4) \% m \Rightarrow idx_4$

► Where:

►  $k_i$  and  $idx_i$  are arbitrary

►  $k_0 \neq k_1 \neq k_2 \neq k_3 \neq k_4$

►  $idx_0 \neq idx_1 \neq idx_2 \neq idx_3 \neq idx_4$

$k_1$
$k_4$
$k_0$
$k_2$
$k_3$

Table contains  $n$  entries  
Table has capacity  $m$

$$n \leq m$$

Looking up  $k_4$ :

- Go to  $idx_4$
- Don't need to look any further

➤ Best-case lookup complexity:  $O(1)$

# Load factor

- ▶ Performance of a hash table can be measured by its load factor
- ▶  $\frac{n}{m}$  = load factor
  - ▶  $n$  = number of entries in the table
  - ▶  $m$  = capacity of array (number of buckets)



# Significance of load factor

- ▶ In the best possible layouts, lookup takes  $O(\frac{n}{m})$  complexity
  - ▶ Basically constant time with fixed #entries and table sizes
- ▶ Best layouts occur on average, so average complexity is  $O(\frac{n}{m})$
- ▶ In separate chaining:
  - ▶ Means chains are pretty evenly sized across buckets
- ▶ In linear probing:
  - ▶ Means entries were inserted at the index they hashed to or very close to it
- ▶  $\frac{n}{m}$  could keep growing with separate chaining though

# Constant load factor?

- ▶ With **the best layout**, maintaining a **constant, low load factor** will get us  $O(1)$  *roughly*
- ▶ How can we ensure the layout is close to the best layout?
  - ▶ By using a good hash function
- ▶ How can we ensure that the load factor remains low or constant?
  - ▶ Resize the hash table (increase  $m$ ) whenever the load factor gets too large

# Constant load factor?

- ▶ With **the best layout**, maintaining a constant, low load factor will get us  $O(1)$  *roughly*
- ▶ How can we ensure the layout is close to the best layout?
  - ▶ By using a good hash function
- ▶ How can we ensure that the load factor remains low or constant?
  - ▶ Resize the hash table (increase  $m$ ) whenever the load factor gets too large

# Good hash functions

- ▶ Functions that have uniform distribution in range
- ▶ Function should be deterministic (same output for same input every time)
- ▶ Ideally: probability of each table index after hashing is  $1/m$

# How can we find good hash functions?

- DSSL2 has a way to generate excellent hash functions

```
#lang dssl2 import sbbox_hash

let my_hash_function2 = make_sbox_hash()
my_hash_function2("George")      # => 12954263217209680626
my_hash_function2("Georges")    # => 17471623022190201144

# can create multiple different hash functions
# e.g., for multiple hash tables
# each call to make_sbox_hash generates a new, randomly
# generated (but deterministic!) hash function
let my_hash_function2 = make_sbox_hash()
my_hash_function2("George")      # => 910100432649087933
my_hash_function2("Georges")    # => 16264795133863970133
```



# Constant load factor?

- ▶ With the best layout, maintaining a **constant, low load factor** will get us  $O(1)$  *roughly*
- ▶ How can we ensure the layout is close to the best layout?
  - ▶ By using a good hash function
- ▶ **How can we ensure that the load factor remains low or constant?**
  - ▶ **Resize the hash table (increase  $m$ ) whenever the load factor gets too large**

# Maintaining a reasonable load factor

- ▶ We want the load factor to always stay reasonable, below a threshold, and generally constant
- ▶ How is that possible if we have to keep adding elements to a hash table?
  - ▶ When the load factor reaches a threshold:
    1. Double the size of the array
    2. Rehash the elements

# Why double the array?

- ▶ Doubling the array whenever it gets full means:
  - ▶ We have to resize fewer and fewer times as  $n$  gets larger
  - ▶ Cost of resizing more frequently when  $n$  is smaller, is small
- ▶ Overall cost becomes  $O(1)$  amortized
  - ▶ Same idea can be applied to dynamic arrays implemented with vectors
  - ▶ Amortized analysis later in the quarter!

# Dictionary implementations so far

	Unsorted array of (key, value) pairs	Sorted array of (key, value) pairs; sorted by key	Linked list of (key, value) pairs	Hash table
<b>Lookup</b>	$O(n)$	$O(\log n)$	$O(n)$	$O(1)$ avg + amortized
<b>Insert</b>	$O(n)$	$O(n)$	$O(n)$	$O(1)$ avg + amortized
<b>Delete</b>	$O(n)$	$O(n)$	$O(n)$	$O(1)$ avg + amortized
<b>Member?</b>	$O(n)$	$O(\log n)$	$O(n)$	$O(1)$ avg + amortized