CSE 332 Autumn 2026
Lecture 10: Hashing, pt. 1

Ruth Anderson

http://www.cs.uw.edu/332

Announcements

- “Meet the staff” activity – Come meet us in office hours!
- Participate in Section, earn a late day (see Ed Board post)
- Section Tomorrow: AVL Trees!
- **Conceptual Office Hours**: 10-11am Thursday in CSE2 128. A space to ask about **course content and topics only** *as opposed to direct help with exercises*.
- Exercise 3 Recurrences: Due Fri Oct 23
- Exercise 4 AVL Trees: Due Friday Oct 30
- Midterm: 5-6pm Monday November 9

Dictionary (Map) ADT - Ordered

- Contents:
  - Sets of key+value pairs
  - Keys must be comparable
- Operations:
  - insert(key, value)
    - Adds the (key,value) pair into the dictionary
    - If the key already has a value, overwrite the old value
      - Consequence: Keys cannot be repeated
  - find(key)
    - Returns the value associated with the given key
  - delete(key)
    - Remove the key (and its associated value)

Dictionary Data Structures

Data Structure	Time to insert	Time to find	Time to delete
Unsorted Array	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Unsorted Linked List	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Sorted Array	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$
Sorted Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Heap	$\Theta(\log n)$	$\Theta(n)$	$\Theta(n)$
Binary Search Tree	$\Theta(\text{height})$	$\Theta(\text{height})$	$\Theta(\text{height})$
AVL Tree	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

BSTs and AVL Trees

- Binary Search Tree:
  - A binary tree where for each node, all keys in its left subtree are smaller and all keys in its right subtree are larger
  - Find:
    - If it matches, return the value.
    - If the search key is less than the current node, look left. If it's greater, look right.
    - If we reach an empty spot, find was unsuccessful
  - Insert:
    - Do a find, if it was successful then update the value
    - If it was unsuccessful, add a new node to the empty spot we found.
  - Delete:
    - If the deleted node is a leaf, just remove it
    - If the deleted node had one child, replace it with that one child
    - If the deleted node had 2 children, replace it with the largest key to the left
- AVL Tree:
  - A binary search tree where for each node, the height of its left subtree and the height of its right subtree are off by at most 1.
  - Find:
    - Same as BST
  - Insert:
    - Do a BST insert, then rotate if tree is unbalanced (apply one LL, RR, LR, RL case)
  - Delete: (Not covered this quarter)
    - Do a BST delete, then rotate if the tree is unbalanced (apply LL, RR, LR, RL cases as needed from leaf to root)

Aside: Other Tree-based Dictionaries

We will not cover these, but may be of interest:

- Red-Black Trees
  - Similar to AVL Trees in that we add shape rules to BSTs
  - More “relaxed” shape than an AVL Tree
    - Trees can be taller (though not asymptotically so)
    - Needs to move nodes less frequently
  - This is what Java’s TreeMap uses!
- Tries
  - Similar to a Huffman Tree
  - Requires keys to be sequences (e.g. Strings)
  - Combines shared prefixes among keys to save space
  - Often used for text-based searches
    - Web search, Genomes

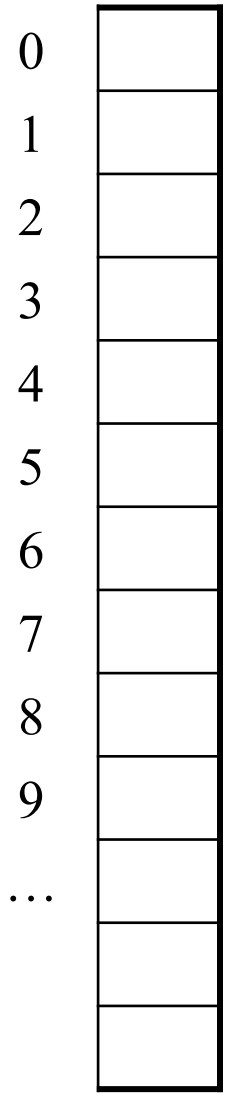
Next topic: Hash Tables

Data Structure	Time to insert	Time to find	Time to delete
Unsorted Array	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Unsorted Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Sorted Array	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$
Sorted Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Binary Search Tree	$\Theta(\text{height})$	$\Theta(\text{height})$	$\Theta(\text{height})$
AVL Tree	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
Hash Table (Worst case)	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Hash Table (Average)	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

Dictionary (Map) ADT - Unordered

- Contents:
  - Sets of key+value pairs
  - ~~Keys must be comparable~~ Keys have a hash function
- Operations:
  - insert(key, value)
    - Adds the (key,value) pair into the dictionary
    - If the key already has a value, overwrite the old value
      - Consequence: Keys cannot be repeated
  - find(key)
    - Returns the value associated with the given key
  - delete(key)
    - Remove the key (and its associated value)

Another Dictionary: “Really Big Array”



“Really Big Array” Operations

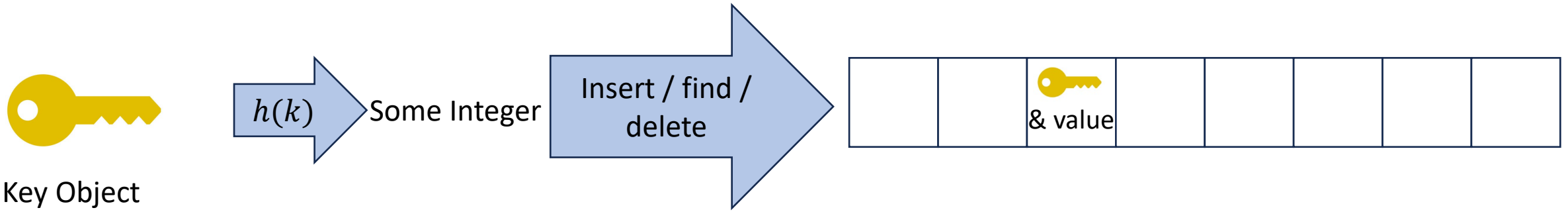
- Think of every key as a number
- Give each key its own index in an array

```
insert(key, value){
    arr[key] = value;
}
find(key){
    return arr[key];
}
delete(key){
    arr[key] = null;
}
```

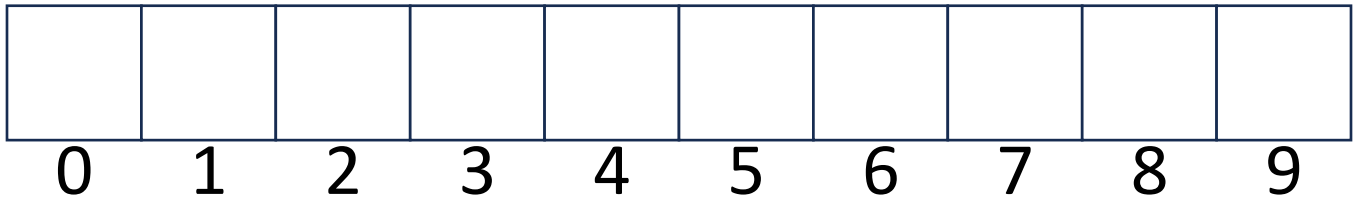
Problems?

Hash Tables

- Idea:
  - Have a small array to store information
  - Use a **hash function** to convert the key into an integer
    - Hash function should "scatter" the keys, behave as if it randomly assigned keys to integers
  - Mod integer by tablesize and Store key (& value) at given index
  - Do something if two keys map to the same place (should be very rare)
    - Collision resolution



Caller-ID Example



- **Key**: Phone Number (as a string)
- **Value**: Person associated with that phone number
- Table size: 10

Sample Hash function:

- $h(phone)$ returns the phone number as an integer
  - $h(\text{"2068675309"}) = 2068675309$
- Store at index: $(\text{Result of hash function}) \ \% \ \text{Table size}$
  - The key “2068675309” is stored at index 9, because 2068675309 % 10 is 9

What Influences Running time?

- How long hashing itself takes
- Likelihood of collisions
  - “quality” of our hash function
  - Size of the array vs number of values in the array
- What we do when we have a collision

What are the Keys?

- Keys can be as simple as an integer or as complex as a Java object

- If the key is a Java object:
  - Which parts of the object will you use in your hash function?
  - Equal objects must hash the same:
    - If `a.equals(b)` or if `c.compare(a,b) == 0`, then we must require:
      `a.hashCode() == b.hashCode()`
  - If you ever override **`.equals()`**
    - You need to override **`hashCode()`** also in a consistent way

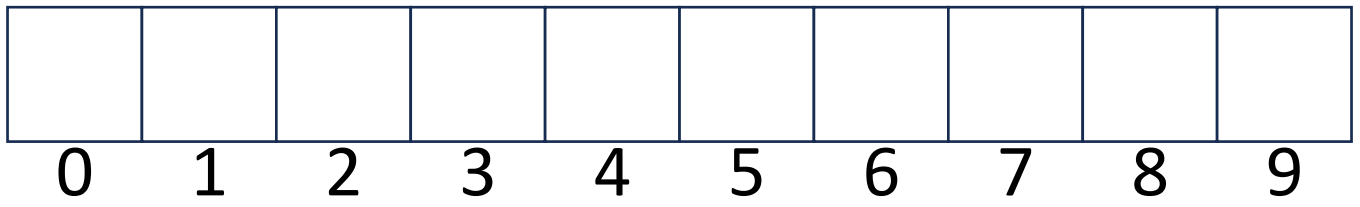
Properties of a “Good” Hash Function

- Definition: A hash function maps objects to integers

- **Consistent**
  - Keys considered “equal” should hash to the same integer
  - Deterministic: running the hash function on the same key twice should yield the same result
- **Uniform**
  - Should be able to use every index in a fixed-size array
  - Should use every index at roughly equal rates
- **Effective**
  - It should be difficult to find two different (unequal) keys which hash to the same integer
  - Given a key, it should be hard to find a different (unequal) key which hashes to the same integer
  - “Avalanche effect”: making a small change to the key yields big changes in the integer it hashes to
- **Efficient**
  - Time to calculate the hash function should be very small

A Bad Hash Function (and phone number trivia)

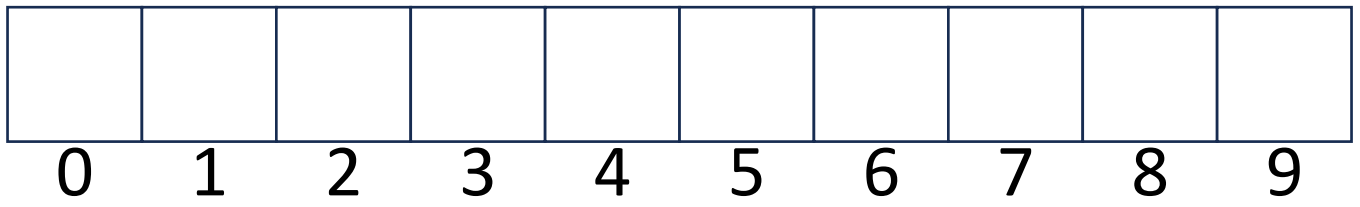
- $h(phone)$ returns ***the first digit*** of the phone number as an integer
  - Assume 10-digit format, with area code first
  - No US phone numbers start with 1 or 0
  - If we're sampling from this class, 2 is by far the most likely
- Consistent?
- Uniform?
- Effective?
- Efficient?



A Bad Hash Function (Answers)

- $h(phone)$ returns ***the first digit*** of the phone number as an integer
  - Assume 10-digit format, with area code first

- Consistent? Yes!
- Uniform? No!
  - No US phone numbers start with 1 or 0
  - If we're sampling from this class, 2 is by far the most likely
- Effective? No!
  - $h(\text{"2068675309"}) = 2$
  - $h(\text{"2068675305"}) = 2$
- Efficient? Yes!



Compare These Hash Functions (for strings)

- Let $s = s_0 s_1 s_2 \ldots s_{m-1}$ be a string of length m
  - Let $a(s_i)$ be the ascii encoding of the character s_i
- $h_1(s) = a(s_0)$
- $h_2(s) = \left(\sum_{i=0}^{m-1} a(s_i)\right)$
- $h_3(s) = \left(\sum_{i=0}^{m-1} a(s_i) \cdot 31^i\right)$
- $h_4(s) = \left(2 \cdot \sum_{i=0}^{m-1} a(s_i) \cdot 31^i\right)$

Properties of Those Example Hash Functions

- Let $s = s_0 s_1 s_2 \ldots s_{m-1}$ be a string of length m
  - Let $a(s_i)$ be the ascii encoding of the character s_i
- $h_1(s) = a(s_0)$
  - Is: consistent, efficient
- $h_2(s) = \left(\sum_{i=0}^{m-1} a(s_i)\right)$
  - Is: consistent, efficient, and possibly uniform
- $h_3(s) = \left(\sum_{i=0}^{m-1} a(s_i) \cdot 31^i\right)$
  - Is: consistent, efficient, uniform, and effective
- $h_4(s) = \left(2 \cdot \sum_{i=0}^{m-1} a(s_i) \cdot 31^i\right)$
  - Is: consistent, efficient, effective

Calculating hash functions quickly

$$h_3(s) = \left(\sum_{i=0}^{m-1} a(s_i) \cdot 31^i\right) =$$

$= s_0*31^0 + s_1*31^1 + s_2*31^2 + s_3*31^3 + \ldots$

- But we need to calculate this quickly…..
- Avoid calculating 31^i directly (don't call `Math.pow()`)

If m = 4, re-write as:

$= s_0 + 31(s_1 + 31(s_2 + 31 * s_3))$

```
for (i = m-1; i >= 0;i--) {
        h = 31*h + s[i];
   }
```

Ideal Insert procedure

Supposing we have a “good” hash function:

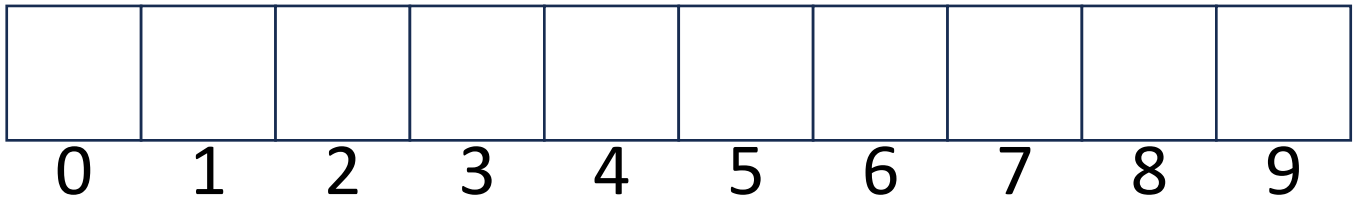
```
insert(key, value) {
     h = key.hash();
     arr[h % table.length] = (key, value);
}
```

Problem: It’s possible that two different keys map to the same index!

This is called a “collision”

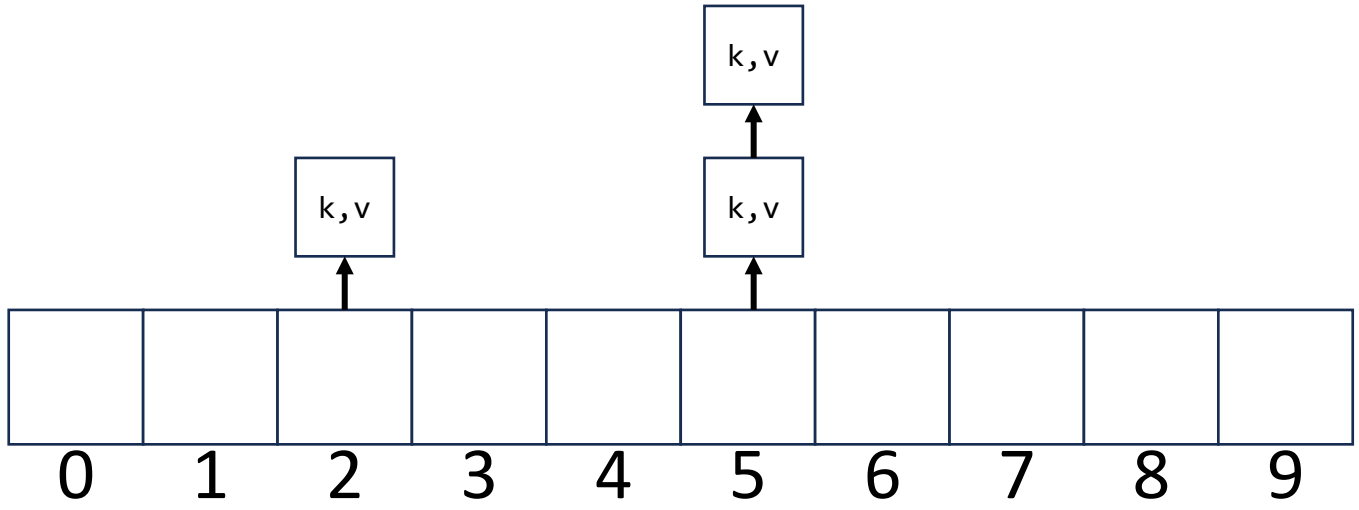
Collision Resolution

- A Collision occurs when we want to insert something into an already-occupied position in the hash table
- 2 main strategies:
  - Separate Chaining
    - Use a secondary data structure to contain the items
      - E.g. each index in the hash table is itself a linked list
  - Open Addressing
    - Use a different spot in the table instead
      - Linear Probing
      - Quadratic Probing
      - Double Hashing



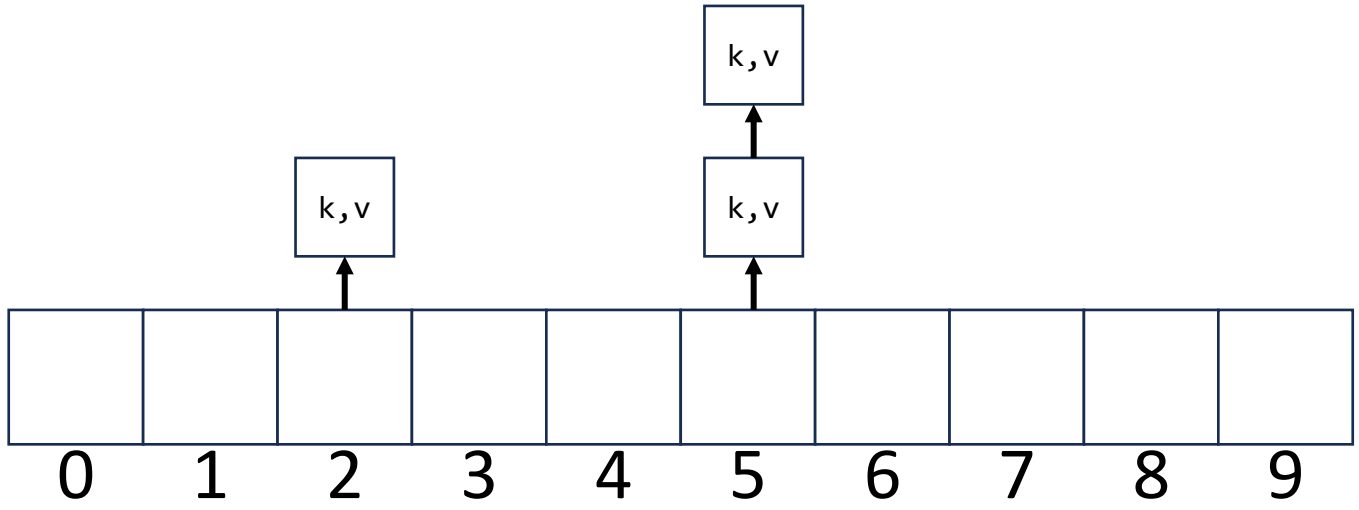
Separate Chaining Insert

- To insert `k, v`:
  - Compute the index using `i = h(k) % table.length`
  - Add the key-value pair to the data structure at `table[i]`



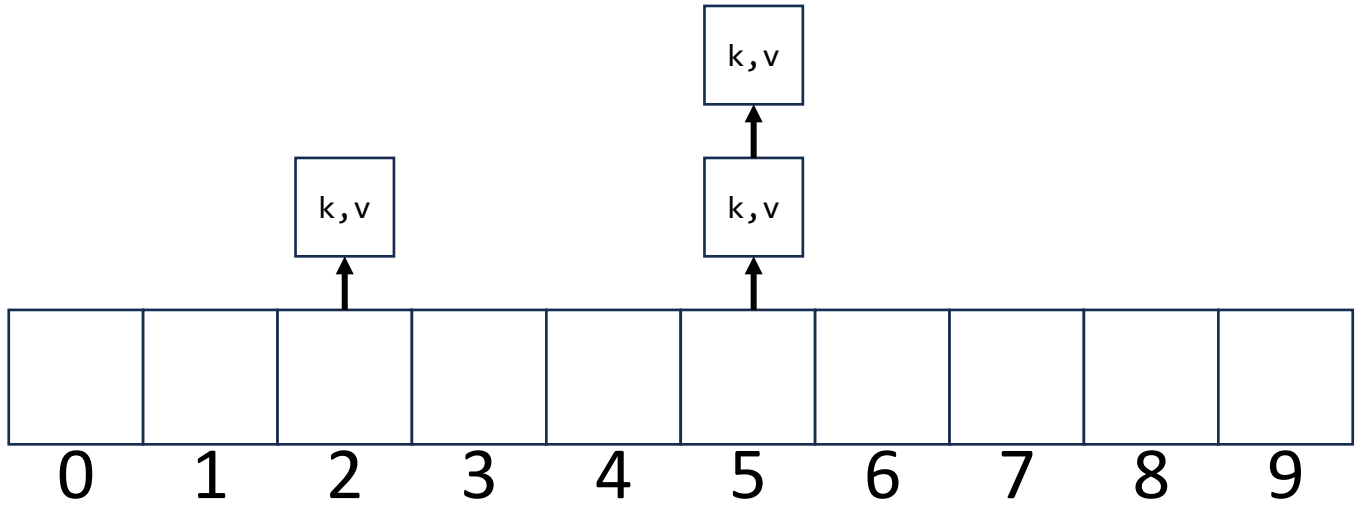
Separate Chaining Find

- To find k:
  - Compute the index using `i = h(k) % table.length`
  - Call find with the key on the data structure at `table[i]`



Separate Chaining Delete

- To delete k:
  - Compute the index using `i = h(k) % table.length`
  - Call delete with the key on the data structure at `table[i]`



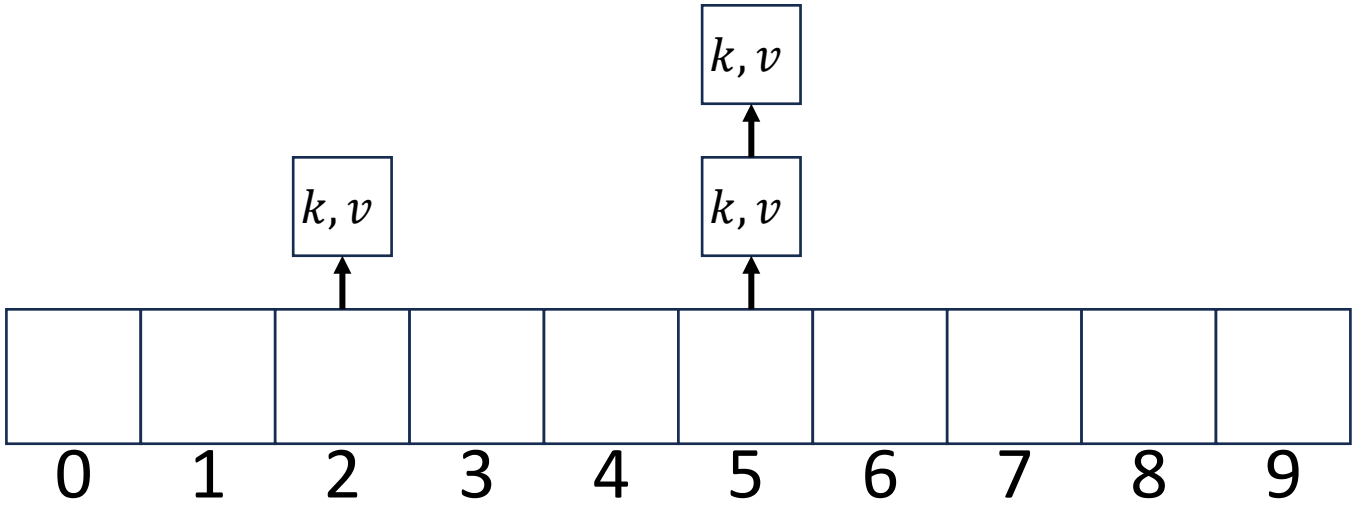
Formal Running Time Analysis

- The **load factor** of a hash table represents the average number of items per "bucket"
  - $\lambda = \frac{n}{length}$
- Assume we have a hash table that uses a linked-list for separate chaining
  - What is the expected number of comparisons needed in an unsuccessful find?
  - What is the expected number of comparisons needed in a successful find?
- How can we make the expected running time $\Theta(1)$?

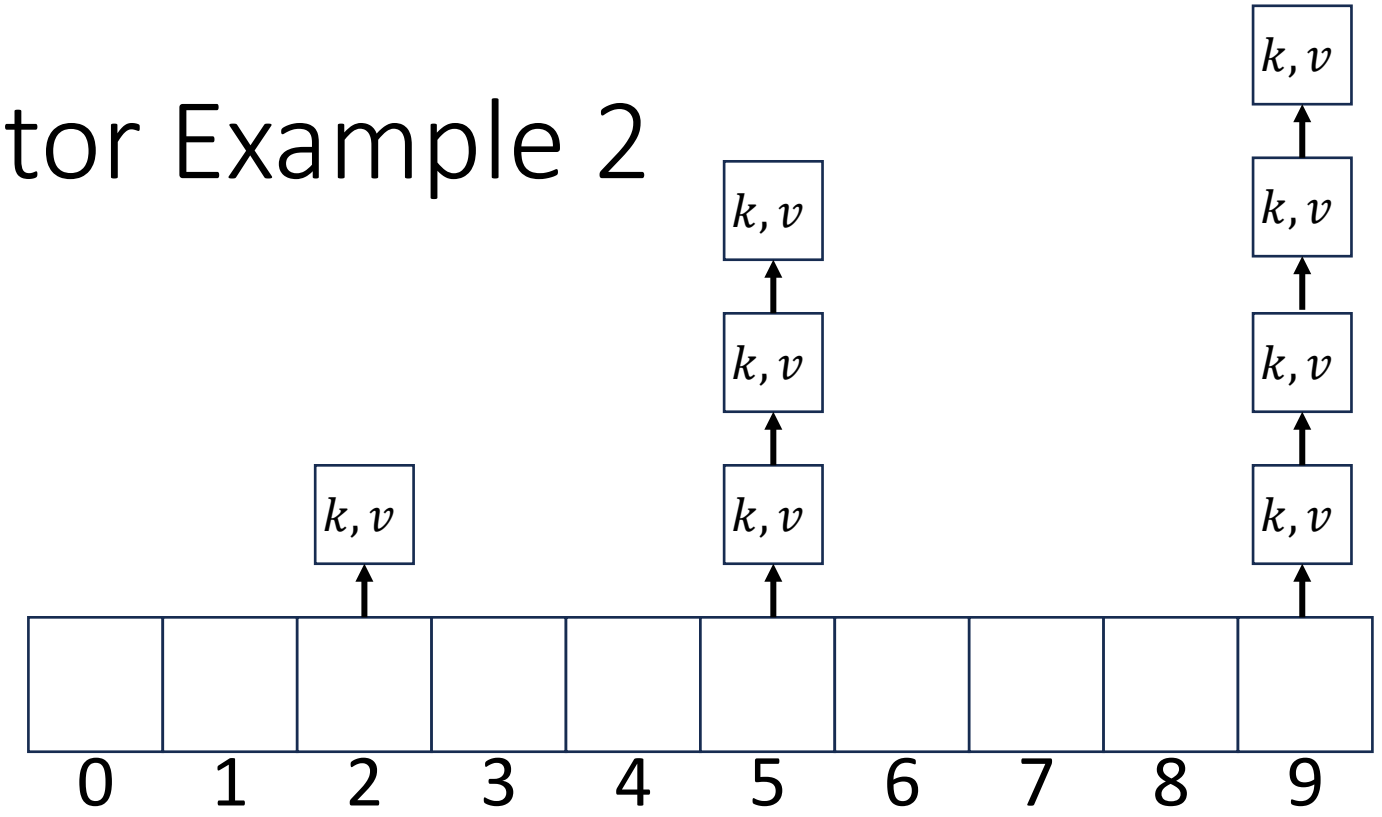
Formal Running Time Analysis (Answers)

- The **load factor** of a hash table represents the average number of items per "bucket"
  - $\lambda = \frac{n}{length}$
- Assume we have a hash table that uses a linked-list for separate chaining
  - What is the expected number of comparisons needed in an unsuccessful find?
    - λ
  - What is the expected number of comparisons needed in a successful find?
    - $\frac{\lambda}{2}$
- How can we make the expected running time $\Theta(1)$?
  - Pick a constant value, resize the array whenever λ exceeds that constant
    - We'll talk about which constant we should pick later

Load Factor Example 1



Load Factor Example 2



Load Factor Example 3

