CSE 332 Autumn 2026
Lecture 11: Hashing, pt. 2

Ruth Anderson

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

Announcements

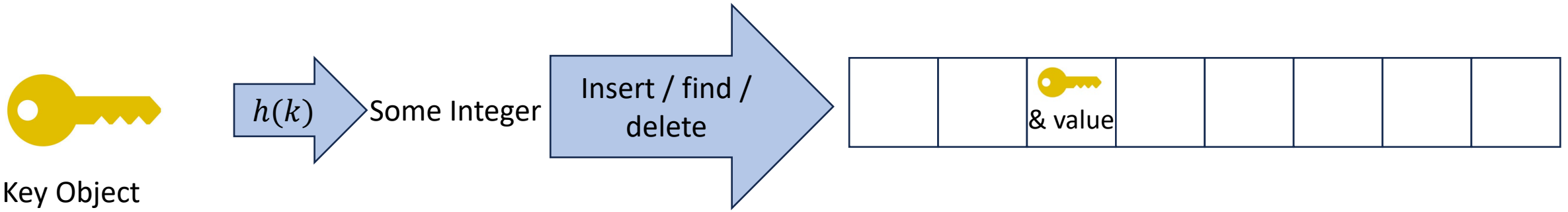
- “Meet the staff” activity – Come meet us in office hours!
- Participate in Section, earn a late day (see Ed Board post)
- Exercise 3 Recurrences: Due TONIGHT, Fri Oct 23
- Exercise 4 AVL Trees: Due Friday Oct 30
- Exercise 5 Hashing: Due Friday November 6
- Midterm: 5-6pm Monday November 9

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)

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



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

Ideal Insert procedure

Supposing we have a “good” hash function:

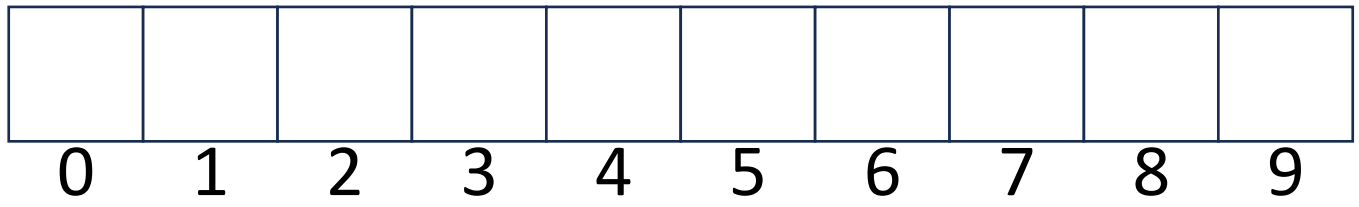
```
insert(key, value) {
     h = key.hash();
     arr[h % table.length] = (key, value);
}
```

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

This is called a “collision”

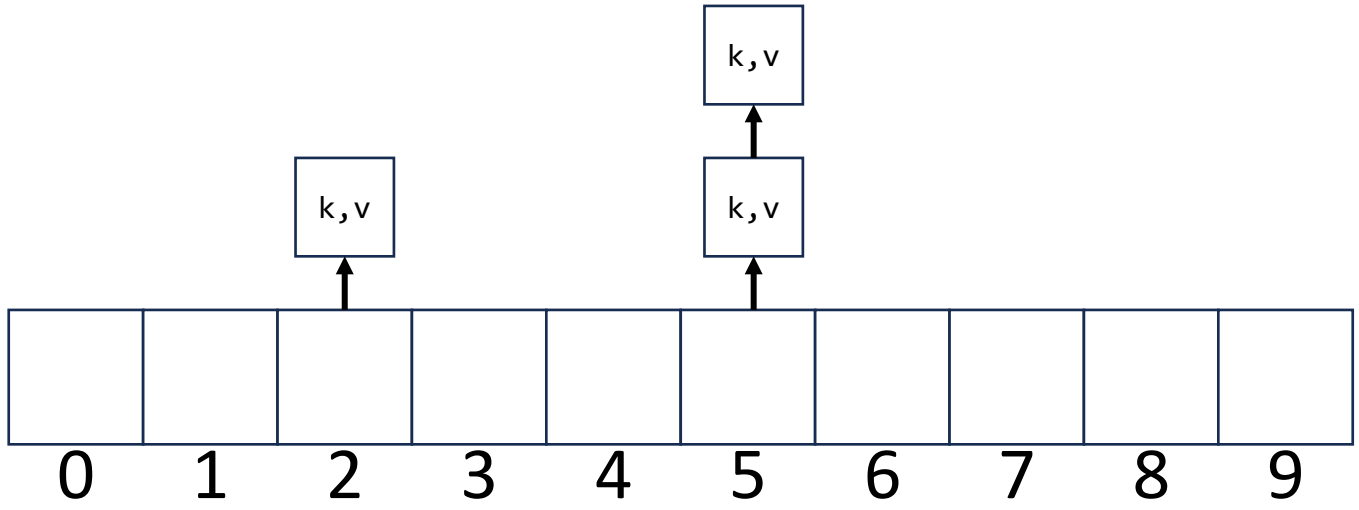
Collision Resolution

- 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



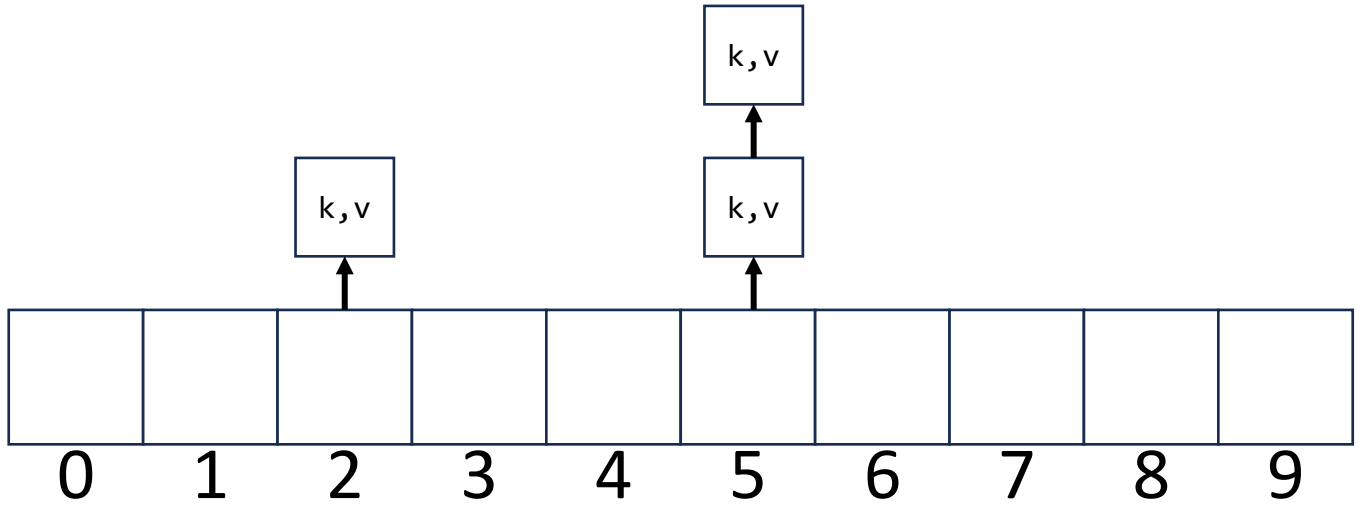
Separate Chaining Insert

- To insert `k, v`:
  - Compute the index using `i = h(k) % table.length`
  - (Call find with the key on the data structure at `table[i]`, no duplicate keys!)
  - If not already present, add the key-value pair to the data structure at `table[i]`



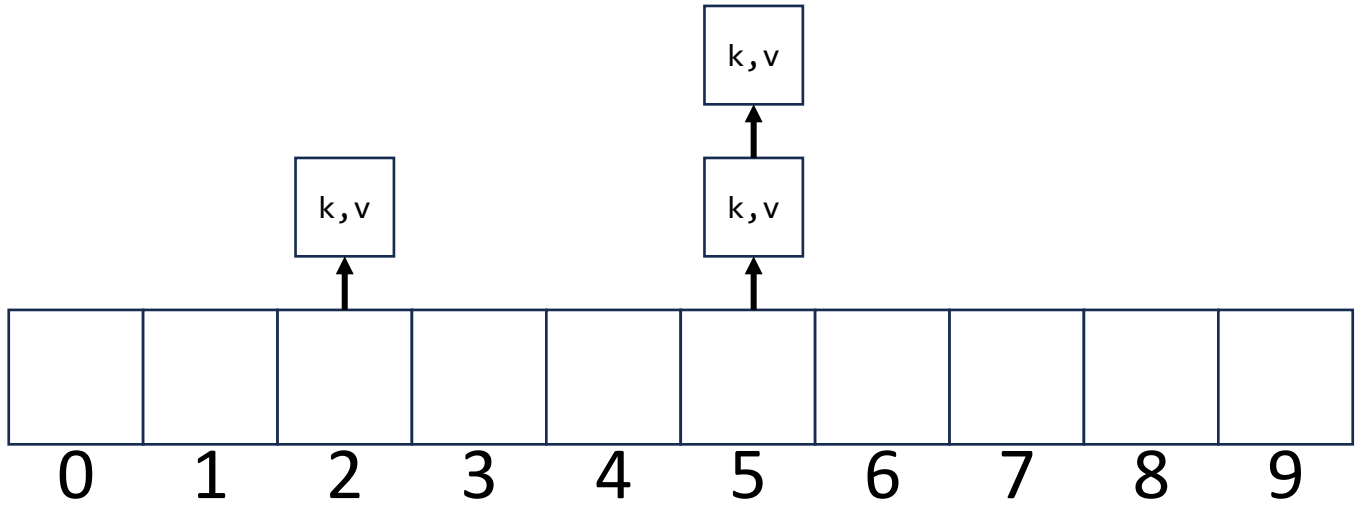
Separate Chaining Find

- 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

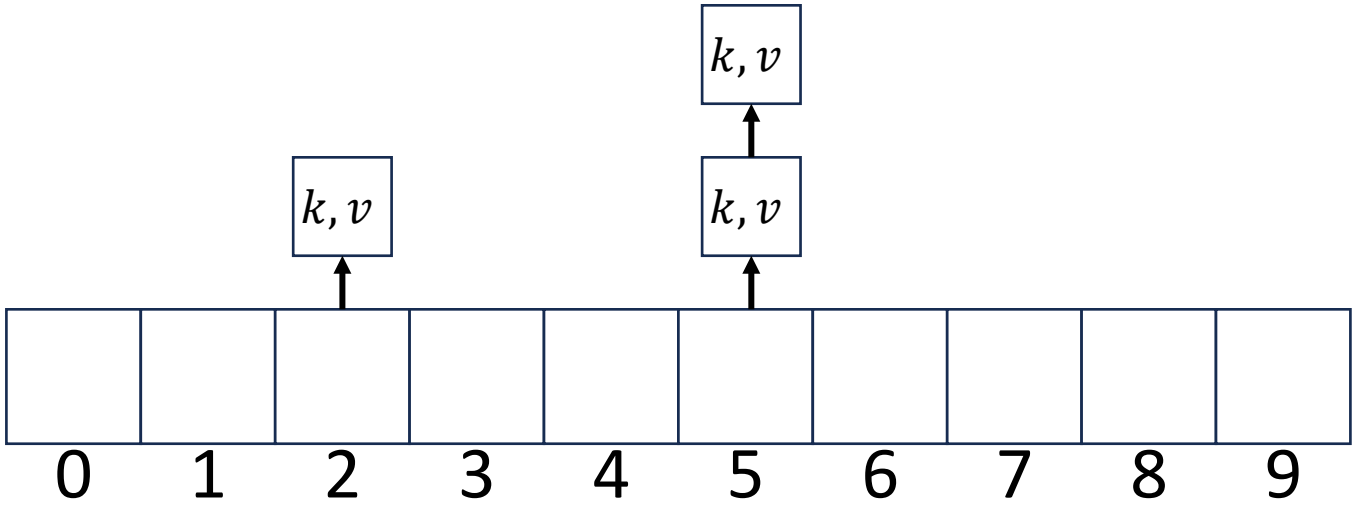
- To delete k:
  - Compute the index using `i = h(k) % table.length`
  - Call delete with the key on the data structure at `table[i]`



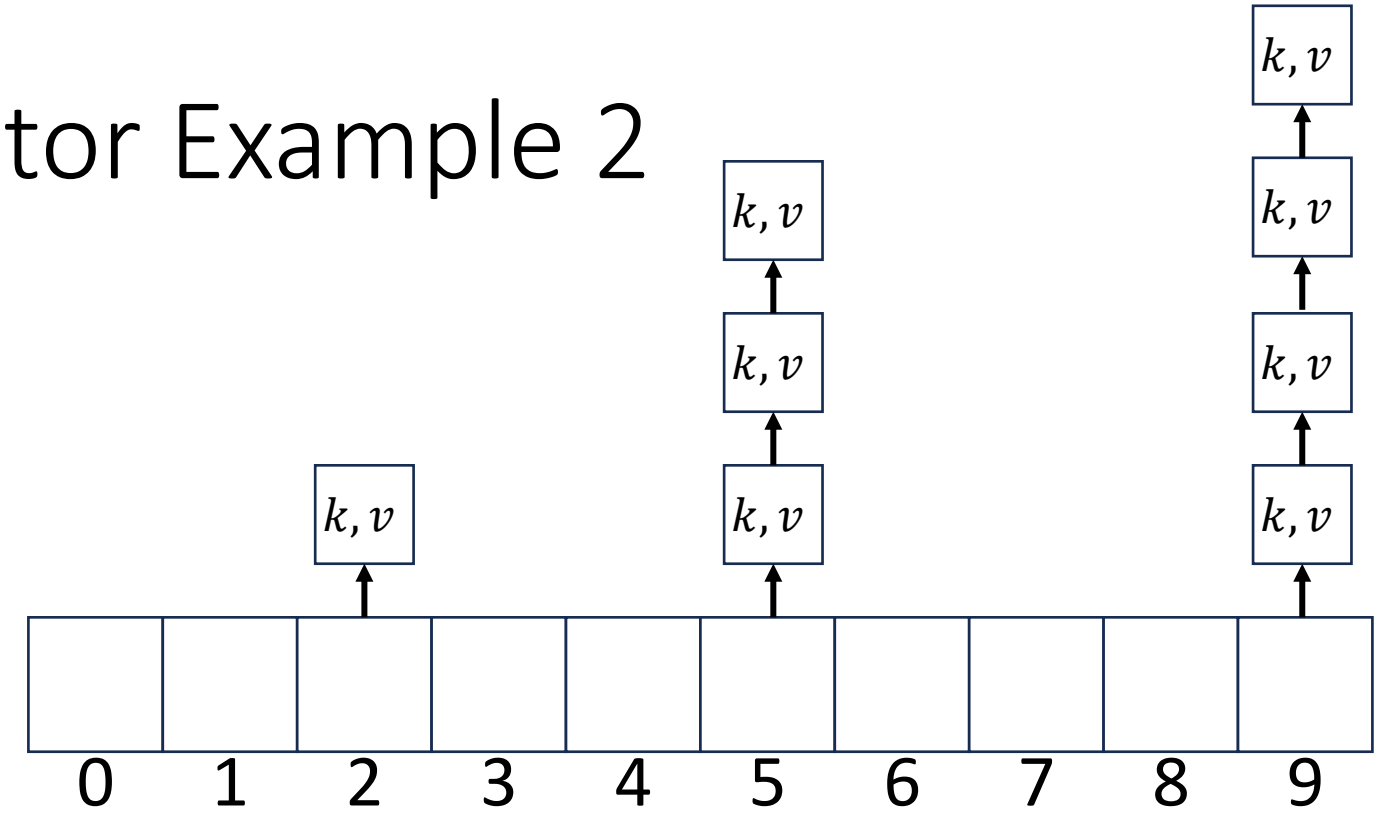
Formal Running Time Analysis

- 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 $\Theta(1)$?

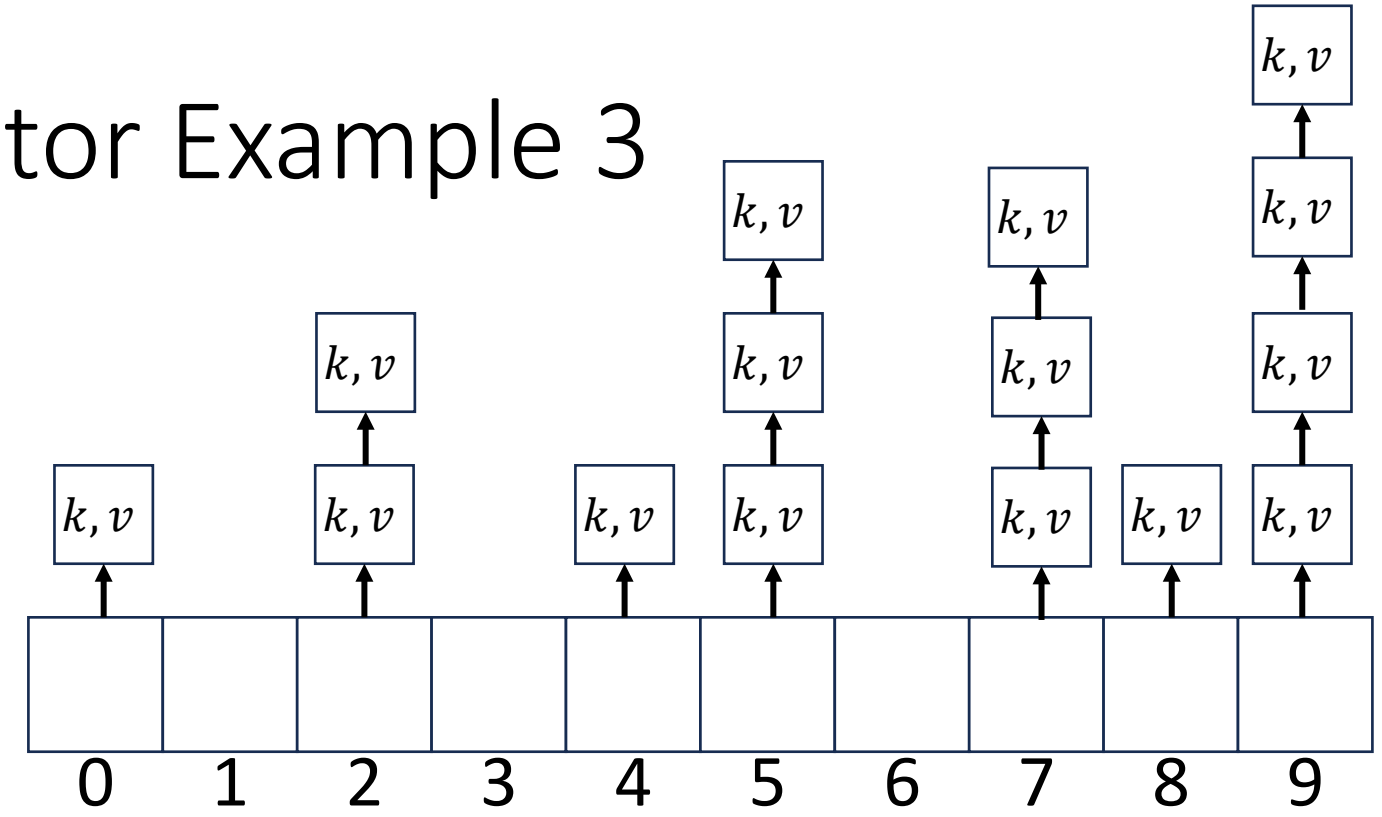
Load Factor Example 1



Load Factor Example 2

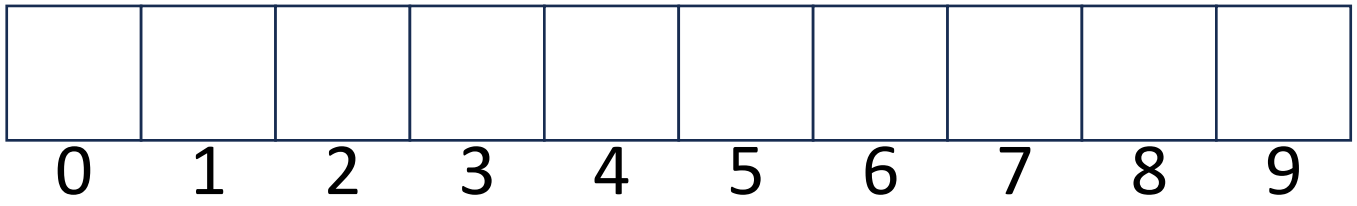


Load Factor Example 3



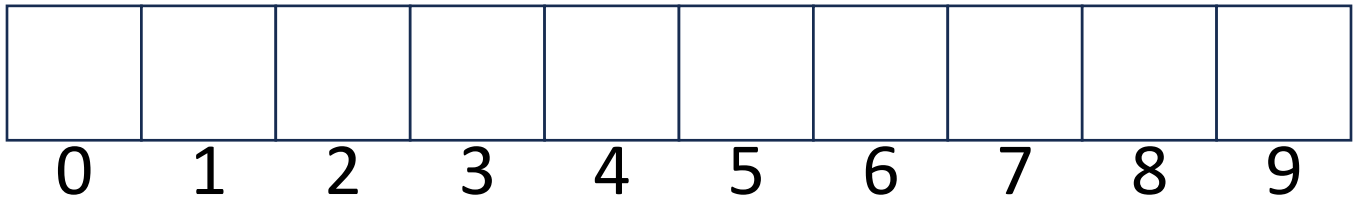
Collision Resolution: Open Addressing

- 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



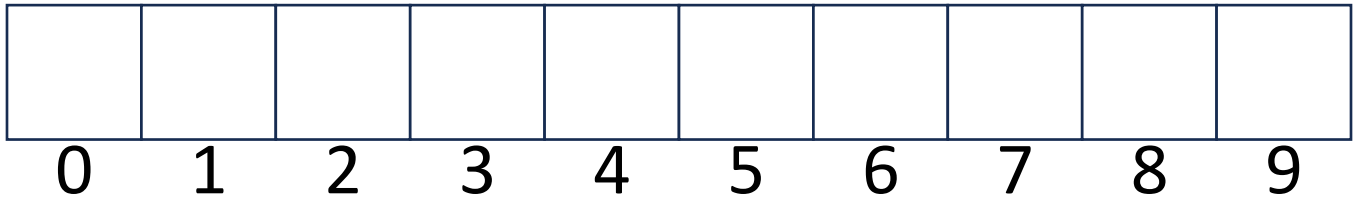
Collision Resolution: Linear Probing

- When there's a collision, use the next open space in the table



Linear Probing: Insert Procedure

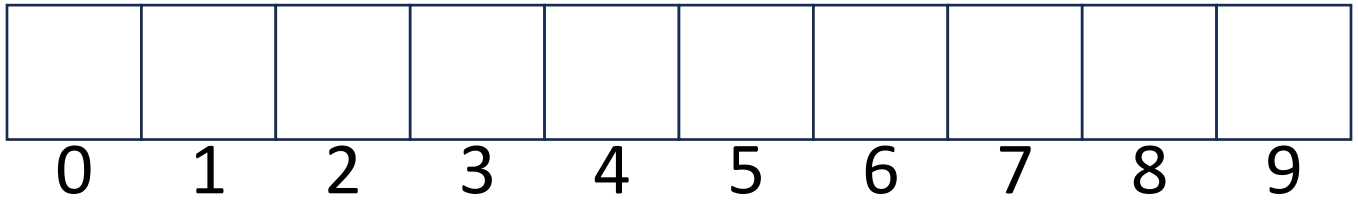
- To insert `k,v`
  - Calculate `i = h(k) % table.length`
  - If `table[i]` is occupied then try index `(i+1) % table.length`
  - If that is occupied try index `(i+2) % table.length`
  - If that is occupied try index `(i+3) % table.length`
  - …



Questions about Linear Probing:

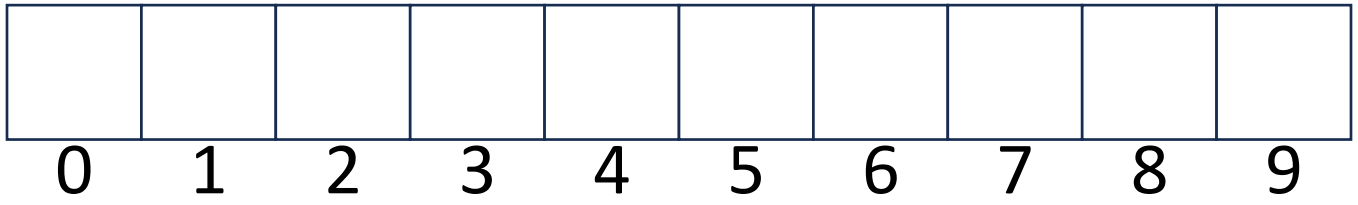
- How to implement find?
  - What if the key is present? Not present?

- How should we implement delete?



Linear Probing: Find

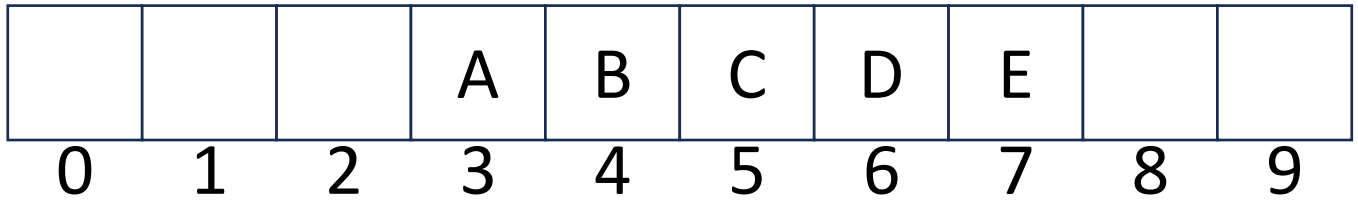
- To find key k
  - Calculate `i = h(k) % table.length`
  - If `table[i]` is occupied but doesn't have k, check `(i+1) % table.length`
  - If that is occupied and doesn't contain k, check `(i+2) % table.length`
  - If that is occupied and doesn't contain k, check `(i+3) % table.length`
  - Repeat until you either find k or else you reach an empty cell in the table



Linear Probing: Delete is Hard (Example 1)

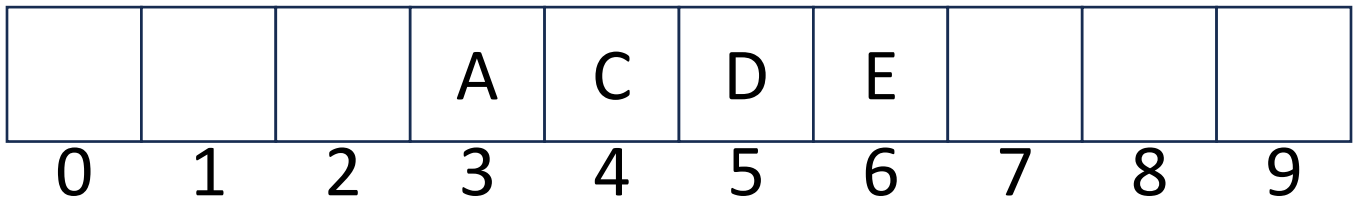
- Suppose we insert A, B, C, D, and E in that order, where all map to 3
- How should we delete B?

Before:



We need future probing to work correctly, so we must fill in the hole.

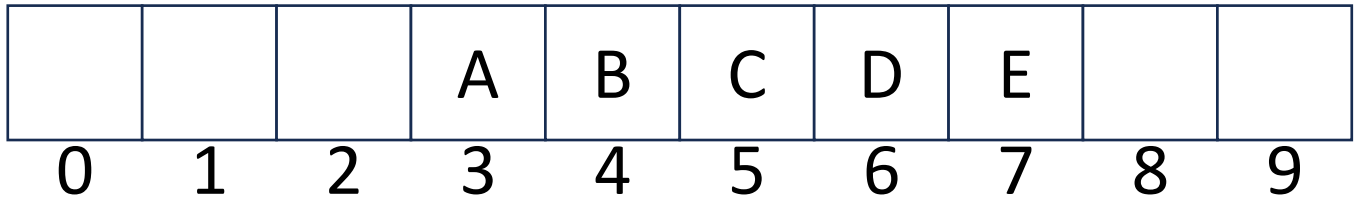
After:



Linear Probing: Delete is Hard (Example 2)

- Suppose we insert A, B, C, D, and E in that order, where A,B,E all map to 3 and C,D map to 5.
- How should we delete B?

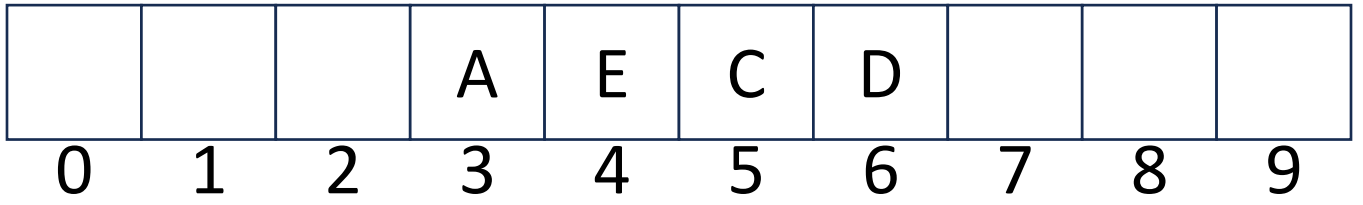
Before:



We need future probing to work correctly, so we must fill in the hole.

We cannot move C or D over because they map to an index later in the probe path

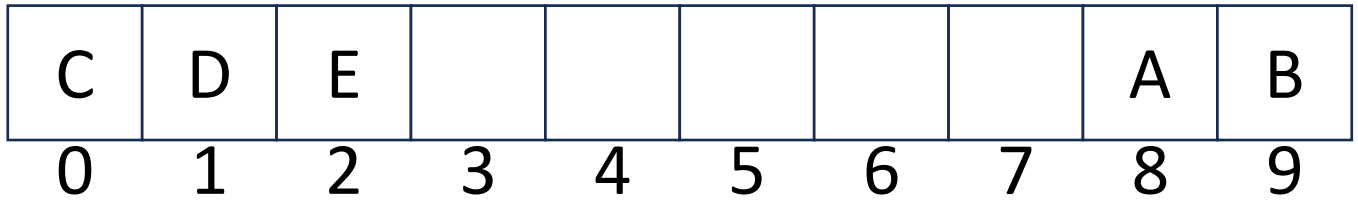
After:



Linear Probing: Delete is Hard (Example 3)

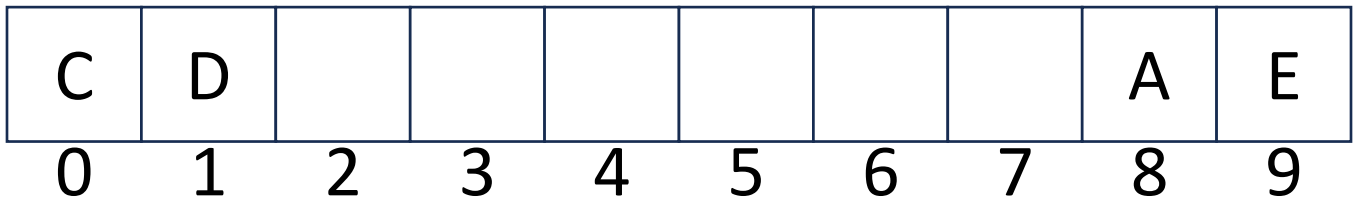
- Suppose we insert A, B, C, D, and E in that order, where A,B,E map to 8 and C,D map to 0.
- How should we delete B?

Before:



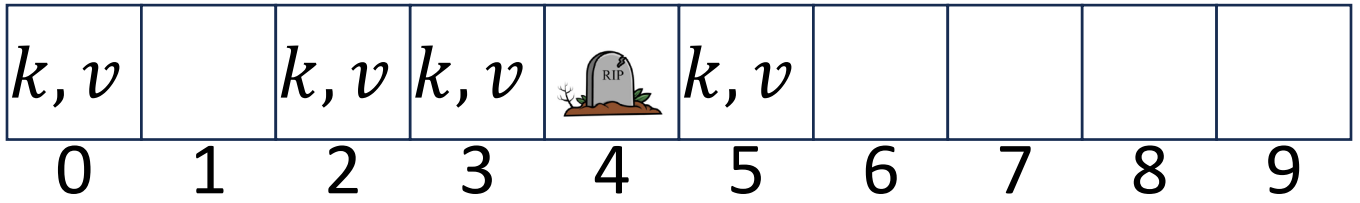
We cannot move C or D over because they map to an index later in the probe path, even though the index number happens to be smaller.
E moves to a larger index because it's earlier in its probe path

After:



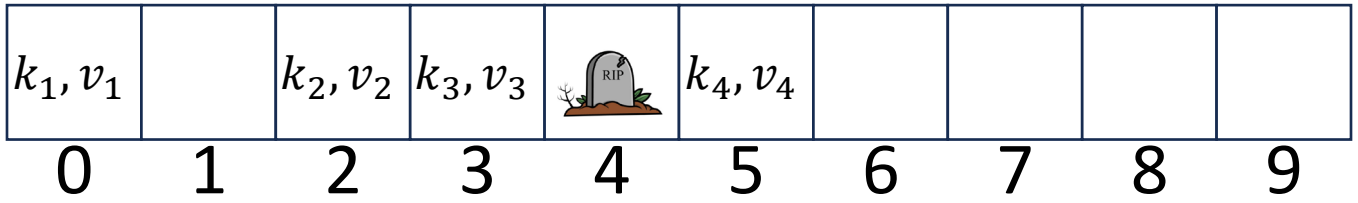
Linear Probing: Delete

- **Option 1 (harder)**: Plug the hole with other items in a way that makes probes behave correctly
  - Something like: to delete at index i, starting from i and going until we find an empty index, if probe path of the key at index j has index i before j then move the key-value pair at j to index i, then repeat on index j.
- **Option 2 (easier)**: "Tombstone" deletion. Leave a special object that indicates an something was deleted from there
  - The tombstone does not act as an open space when finding (so keep looking after its reached)
  - When inserting you can replace a tombstone with a new item



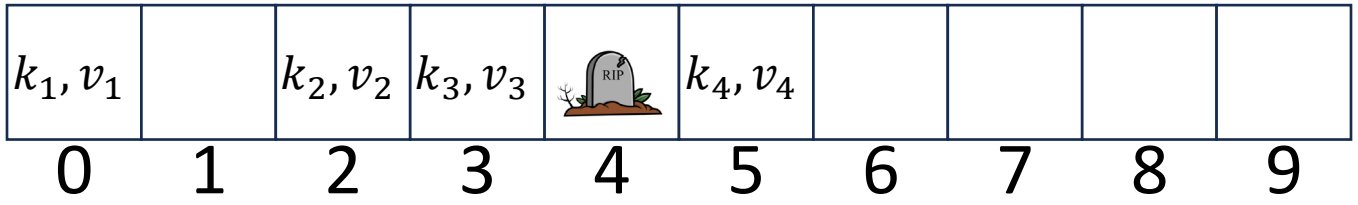
Linear Probing + Tombstone: Find

- To find key k
  - Calculate `i = h(k) % table.length`
  - While `table[i]` has a key other than k, set `i = (i+1) % table.length`
  - If you come across k return `table[i]`
  - If you come across an empty index, the find was unsuccessful
    - Tombstones do not count as empty!



Linear Probing + Tombstone: Insert

- To insert `k,v`
  - Calculate `i = h(k) % table.length`
  - While `table[i]` has a key other than k, set `i = (i+1) % table.length`
    - If `table[i]` has a tombstone, set `dest = i`
      - That is where we will insert if the find is unsuccessful
    - If you come across k, set `table[i] = k,v`
    - If you come across an empty index, the find was unsuccessful
      - Set `table[x] = k,v` if we saw a tombstone
    - Set `table[x] = k,v` otherwise

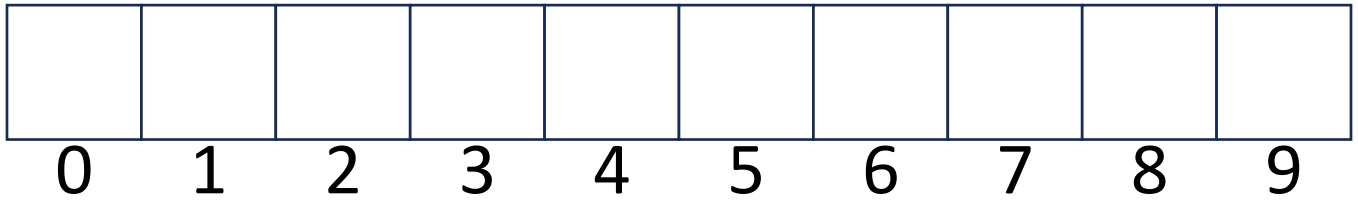


Downsides of Linear Probing

- What happens when λ approaches 1?
  - Get longer and longer contiguous blocks
  - A collision is guaranteed to grow a block
    - Larger blocks experience more collisions
    - Called Primary Clustering
    - Feedback loop!
- What happens when λ exceeds 1?
  - Impossible!
  - You can’t insert more stuff

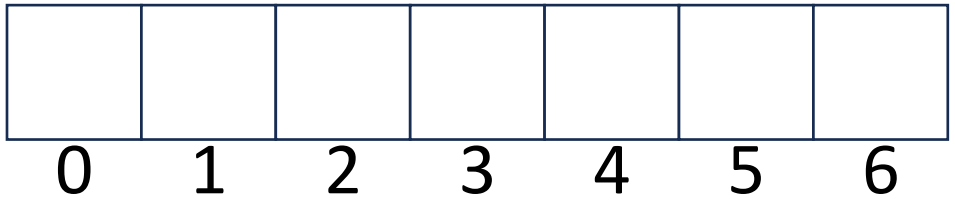
Quadratic Probing: Insert Procedure

- To insert `k,v`
  - Calculate `i = h(k) % table.length`
  - If `table[i]` is occupied then try `(i+1²) % table.length`
  - If that is occupied try `(i+2²) % table.length`
  - If that is occupied try `(i+3²) % table.length`
  - If that is occupied try `(i+4²) % table.length`
  - …



Quadratic Probing: Example

- Insert:
  - 76 (76 % 7 = 6)
  - 40 (40 % 7 = 5)
  - 48 (48 % 7 = 6)
  - 5 (5 % 7 = 5)
  - 55 (55 % 7 = 6)
  - 47 (47 % 7 = 5)



Using Quadratic Probing

- If you probe `table.length` times, you start repeating indices
- If `table.length` is prime and $\lambda < \frac{1}{2}$ then you're guaranteed to find an open spot in at most `table.length`$/2$ probes

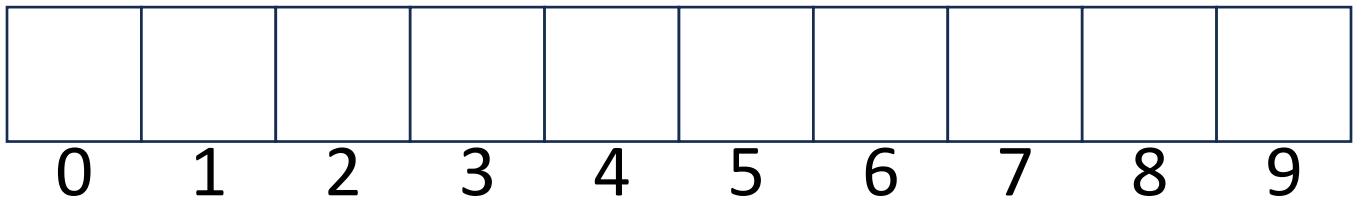
- Helps with the clustering problem of linear probing, but does not help if many things initially hash to the same index…

Clustering reconsidered

- Quadratic probing does not suffer from primary clustering:
  As we resolve collisions we are not merely growing "big blobs" by adding one more item to the end of a cluster, we are looking i^2 locations away, for the next possible spot.
- But quadratic probing does not help resolve collisions between keys that initially hash *to the same **index***
  - Any two keys that initially hash to the same index **will have the same series of moves after that** looking for any empty spot
  - Called secondary clustering
- Can avoid secondary clustering with *a probe function that depends on the key*: double hashing…

Double Hashing: Insert Procedure

- Given h and g are both good hash functions
- To insert `k,v`
  - Calculate `i = h(k) % table.length`
  - If `table[i]` is occupied then try `(i+g(k)) % table.length`
  - If that is occupied try `(i+2*g(k)) % table.length`
  - If that is occupied try `(i+3*g(k)) % table.length`
  - If that is occupied try `(i+4*g(k)) % table.length`
  - …



Summary of Hashing so far

- Separate Chaining is easy
  - **`find, insert, delete`** proportional to load factor on average if using unsorted linked list nodes
  - If using another data structure for buckets (e.g. AVL tree) , runtime is proportional to runtime for that structure.
- Open addressing uses probing, has clustering issues as table fills Why use it:
  - Less memory allocation?
    - Some run-time overhead for allocating linked list (or whatever) nodes; open addressing could be faster
  - Easier data representation?
- Now:
  - Growing the table when it gets too full (aka “rehashing”)

Rehashing

- If your load factor λ gets too large, copy everything over to a larger hash table
  - To do this: make a new, larger array
  - Re-insert all items into the new hash table by reapplying the hash function
    - We need to reapply the hash function because items should map to a different index
  - New array should be "roughly" double the length (but probably still want it to be prime)
- What does "too large" mean?
  - For separate chaining, typically we want $\lambda < 2$
  - For open addressing, typically we want $\lambda < \frac{1}{2}$

Final word on hashing

- The hash table is one of the most important data structures
  - Efficient find, insert, and delete
  - Operations based on sorted order are not so efficient!
  - Useful in many, many real-world applications
  - Popular topic for job interview questions
- Important to use a good hash function
  - Good distribution, Uses enough of key's components
  - Not overly expensive to calculate (bit shifts good!)
- Important to keep hash table at a good size
  - Prime #
  - Preferable λ depends on type of table
- Side-comment: hash functions have uses beyond hash tables
  - Examples: Cryptography, check-sums