

CSE 332 Winter 2026

Lecture 11: hashing 2

Nathan Brunelle

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

Dictionary (Map) ADT

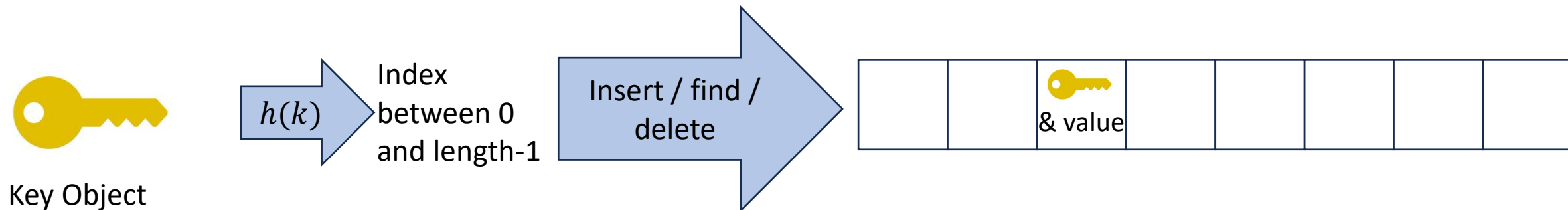
- Contents:
 - Sets of key+value pairs
 - ~~Keys must be comparable~~ Keys have a hash function
- Operations:
 - insert(key, value)
 - Adds the (key,value) pair into the dictionary
 - If the key already has a value, overwrite the old value
 - Consequence: Keys cannot be repeated
 - find(key)
 - Returns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

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)	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$

Hash Tables

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



Properties of a “Good” Hash

- Definition: A hash function maps objects to integers
- **Consistent**
 - Objects considered “equal” should hash to the same value
 - Deterministic: running the hash function on the same object 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 objects which hash to the same value
 - Given an object, it should be hard to find a different object which hashes to the same value
 - “Avalanche effect”: making a small change to the object yields big changes in the value it hashes to
- **Efficient**
 - Time to calculate the hash should be very small

Ideal Insert procedure

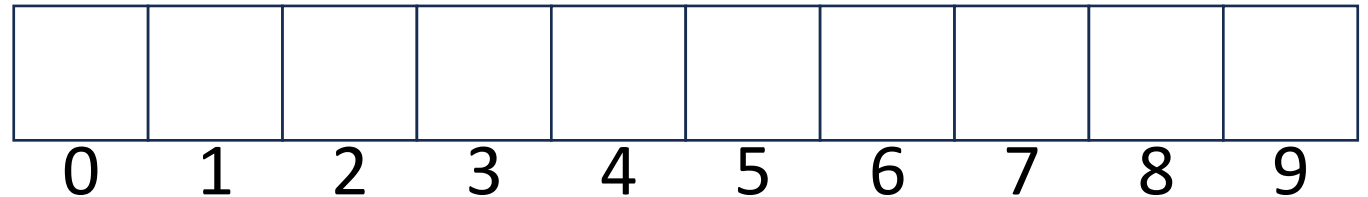
Supposing we have a “good” hash function:

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

Problem: It's possible that two different keys map to the same index!
This is called a “collision”

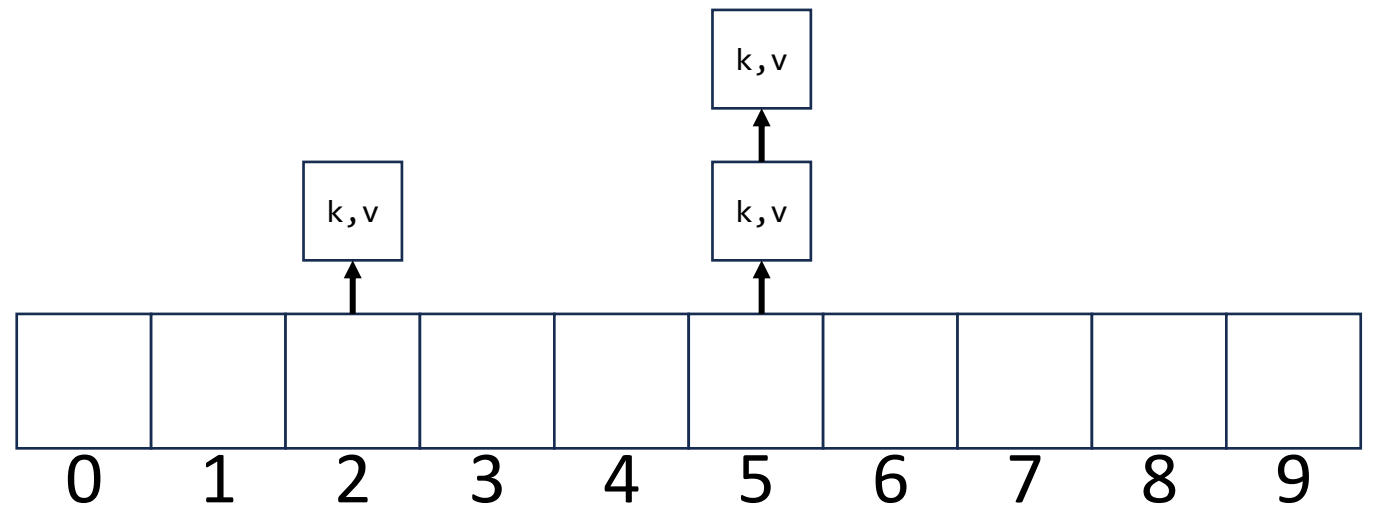
Collision Resolution

- 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



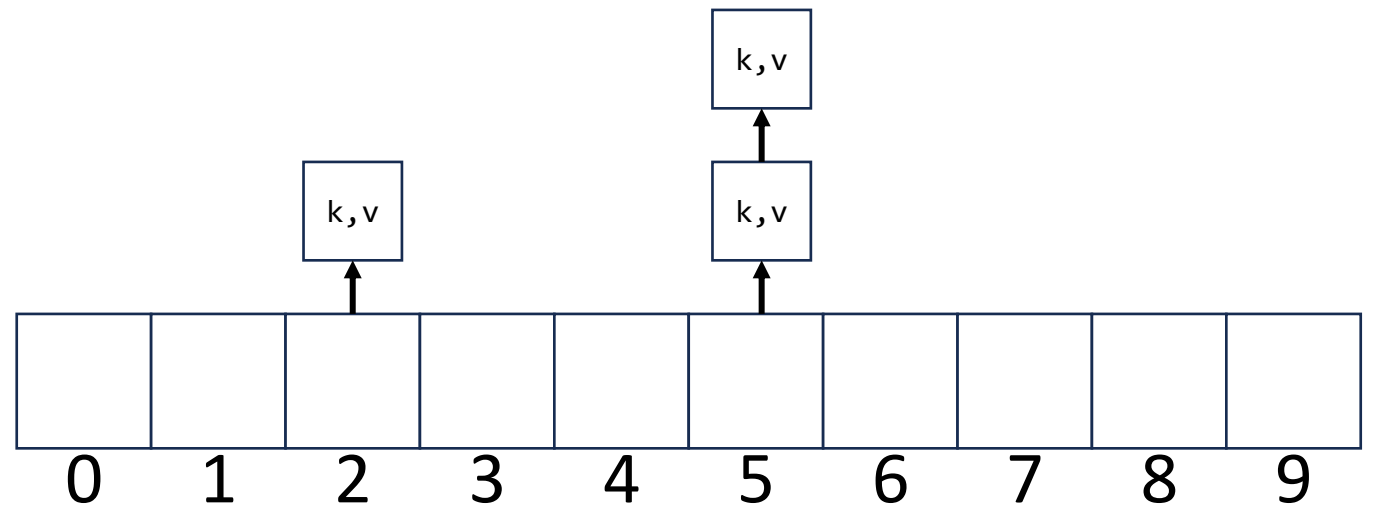
Separate Chaining Insert

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



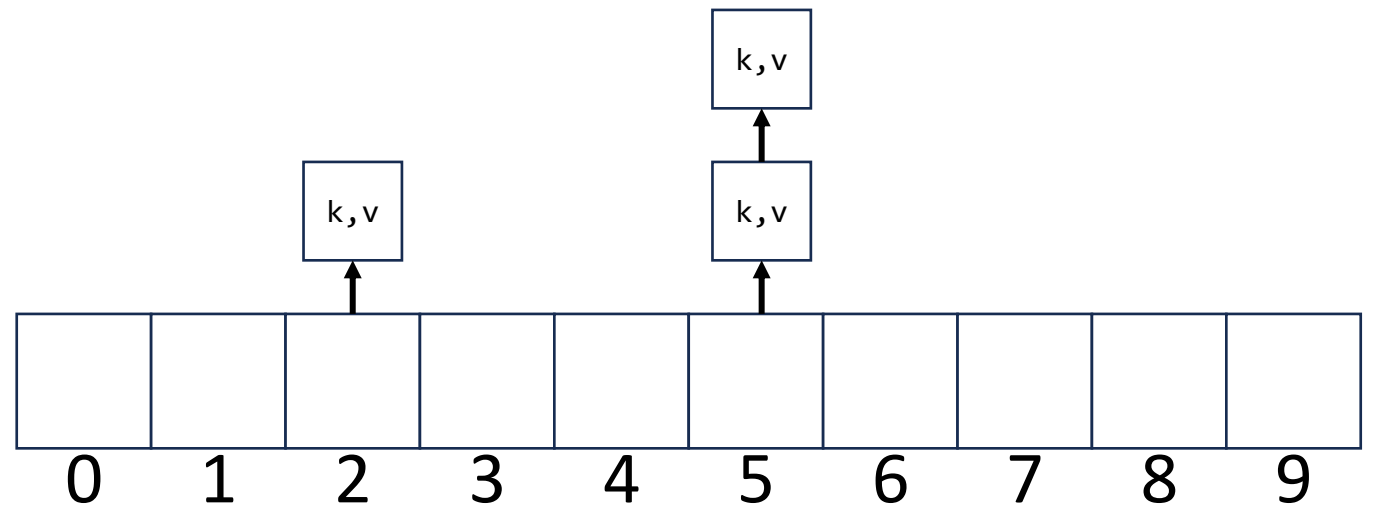
Separate Chaining Find

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



Separate Chaining Delete

- To delete k :
 - Compute the index using $i = h(k) \% \text{table.length}$
 - 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 $\Theta(1)$?

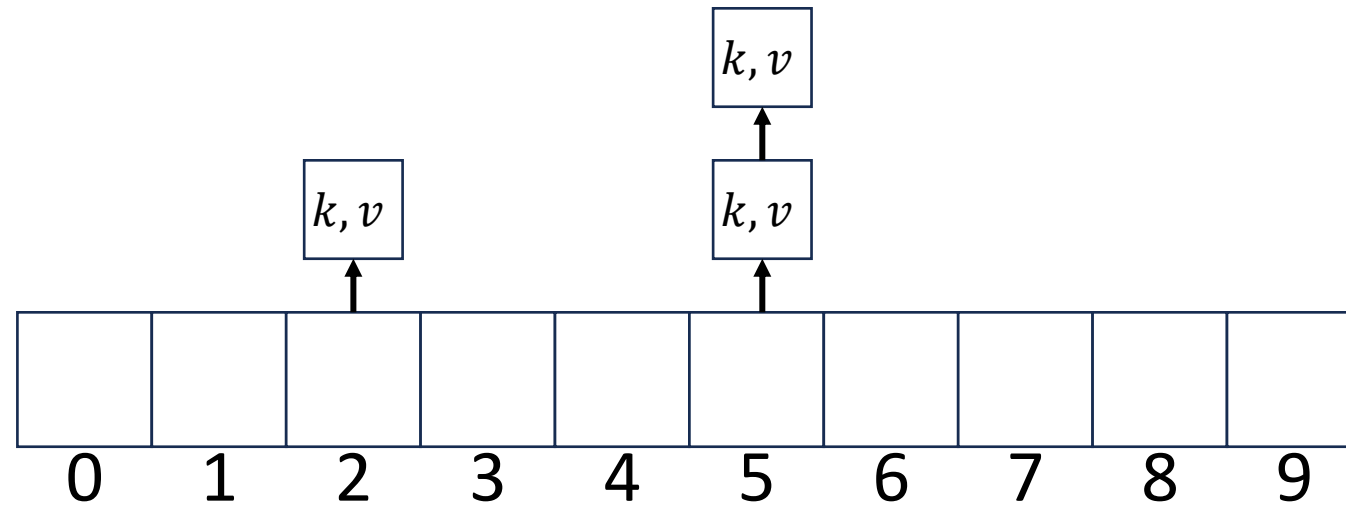
Formal Running Time Analysis

- 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

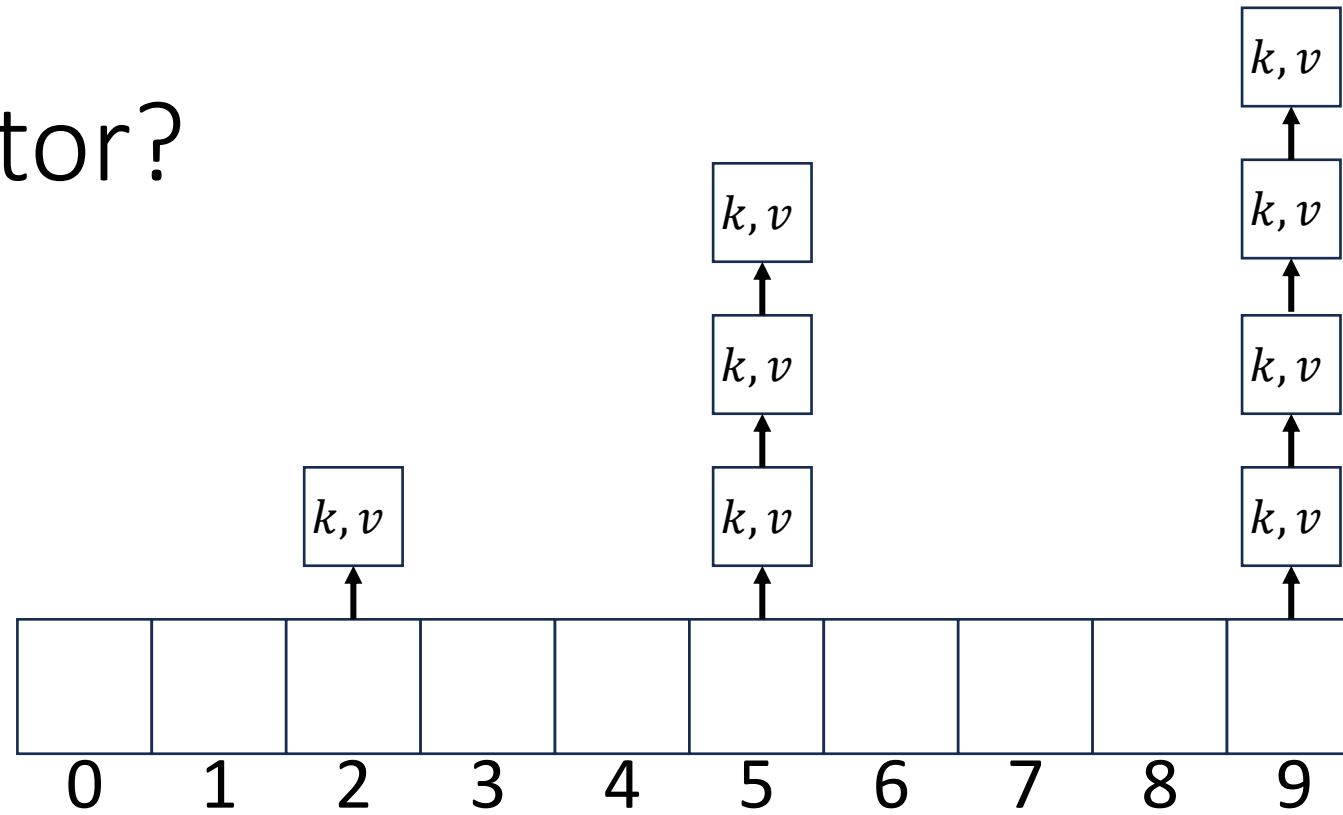
Rehashing

- 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 $\lambda < \frac{1}{2}$

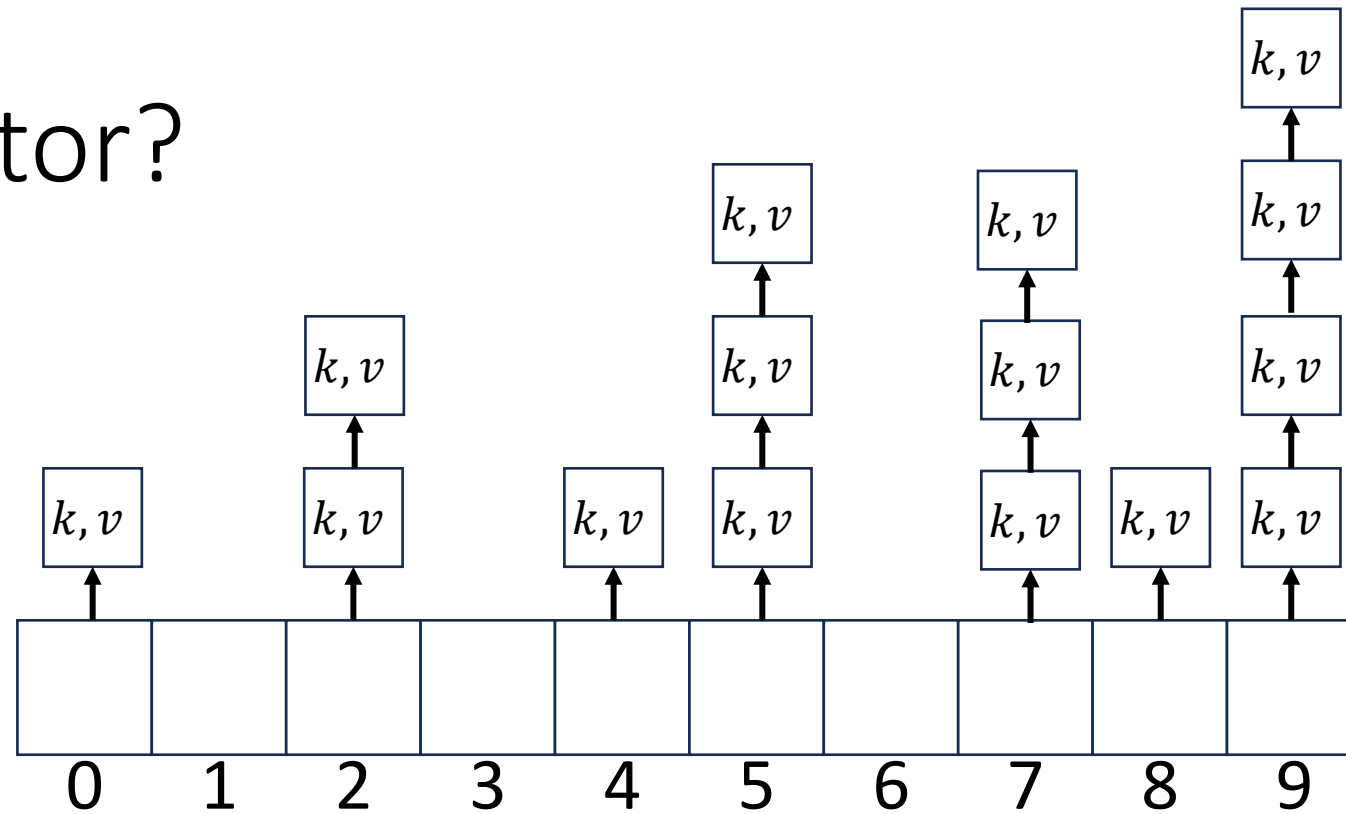
Load Factor?



Load Factor?



Load Factor?



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

- 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

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 $\lambda < \frac{1}{2}$

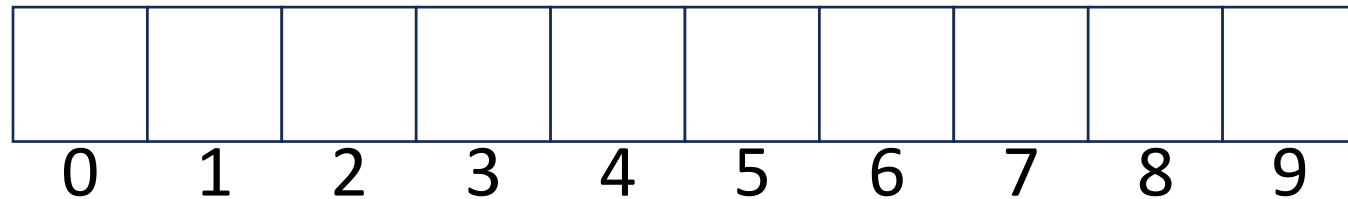
Collision Resolution: Linear Probing

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

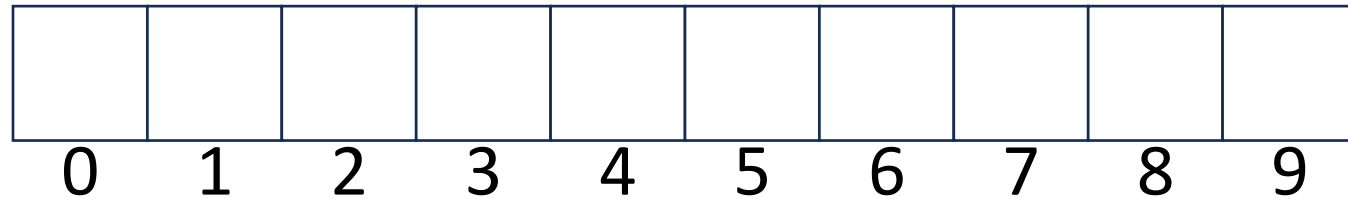
0	1	2	3	4	5	6	7	8	9

Linear Probing: Insert Procedure

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



Linear Probing: Find



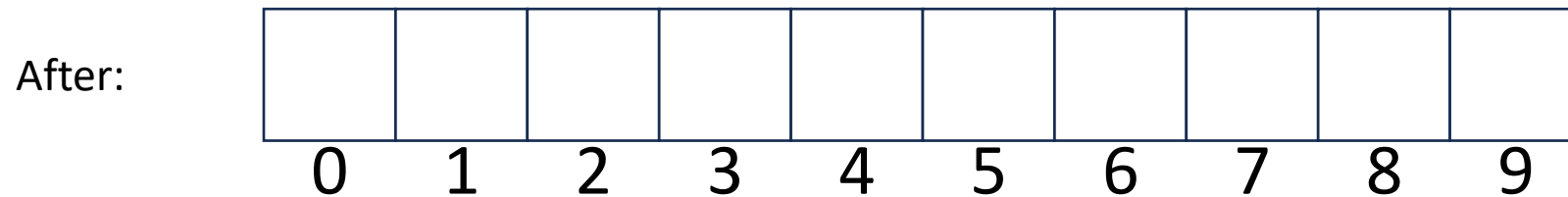
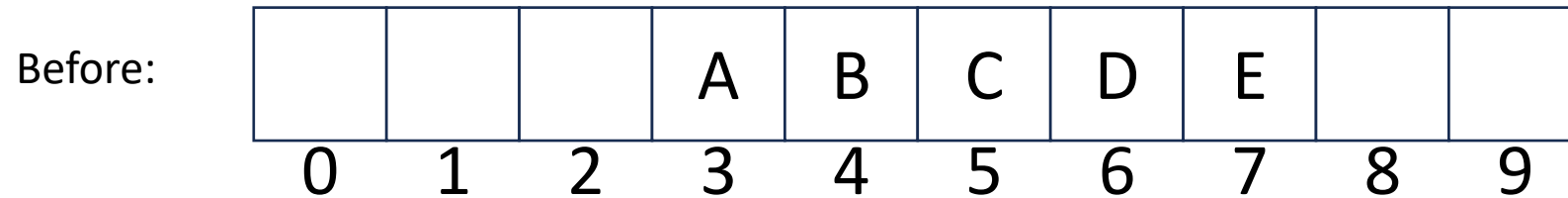
Linear Probing: Find

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

0	1	2	3	4	5	6	7	8	9

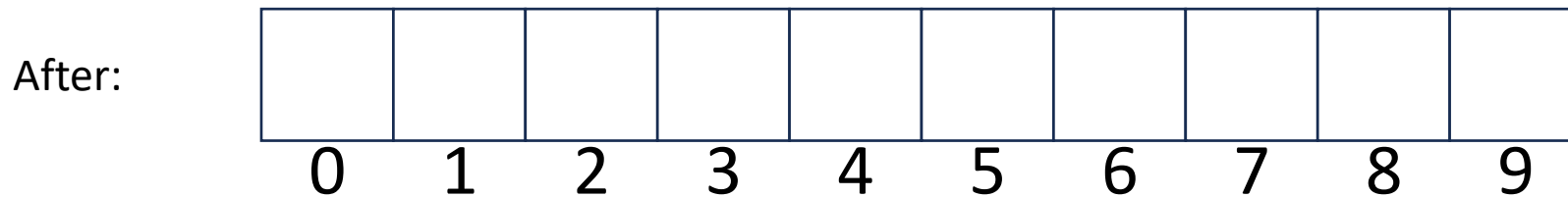
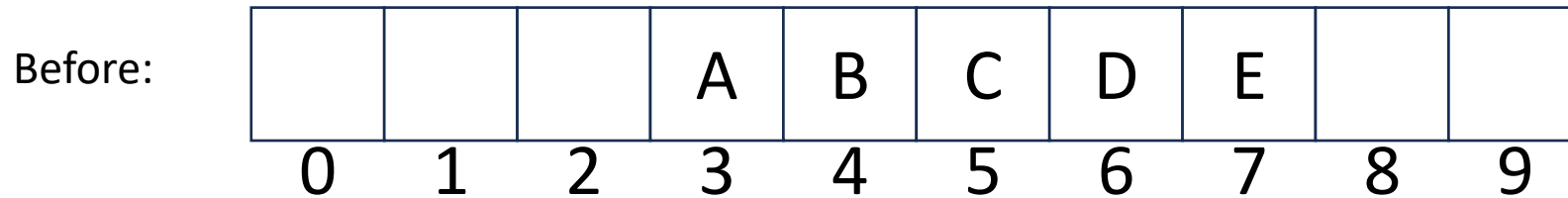
Linear Probing: Delete

- Suppose A, B, C, D, and E all hashed to 3
- Now let's delete B



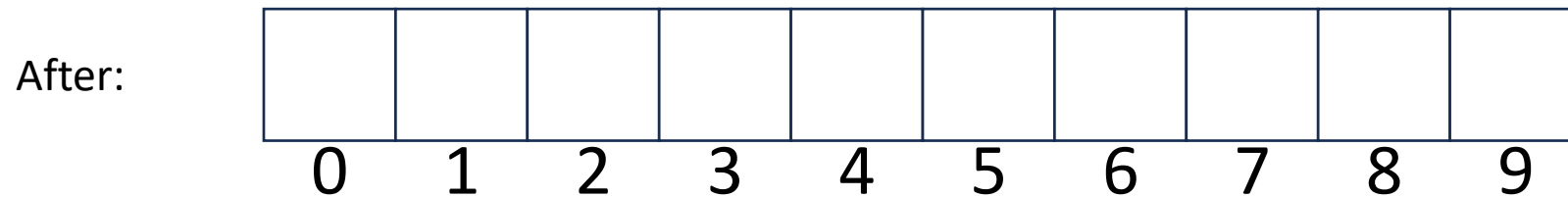
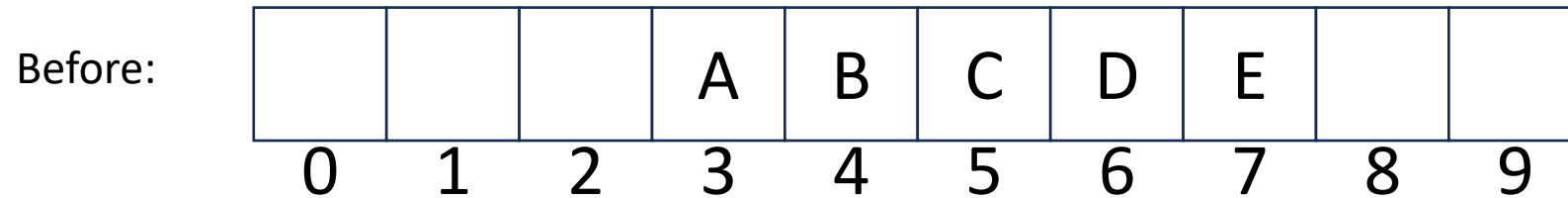
Linear Probing: Delete

- Suppose A, B, and E all hashed to 3, and C and D hashed to 5
- Now let's delete B



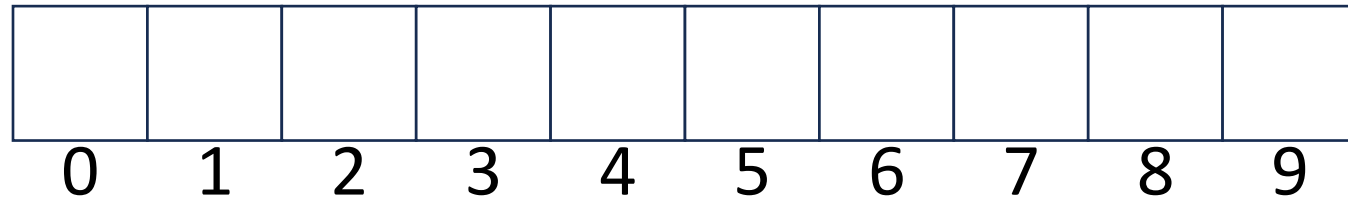
Linear Probing: Delete

- Suppose A and E hashed to 3, and B,C, and D hashed to 4
- Now let's delete B



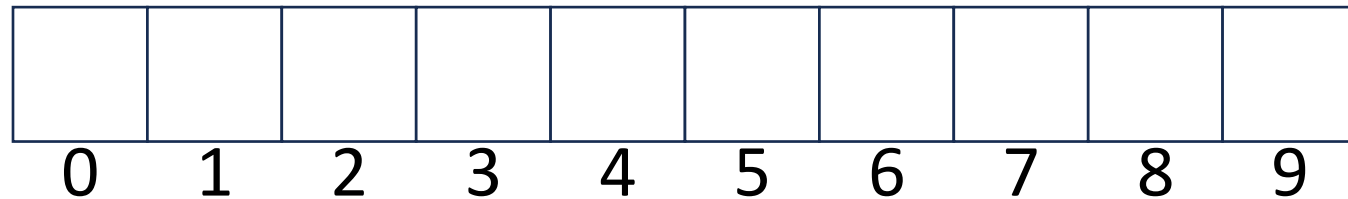
Linear Probing: Delete

- Let's do this together!



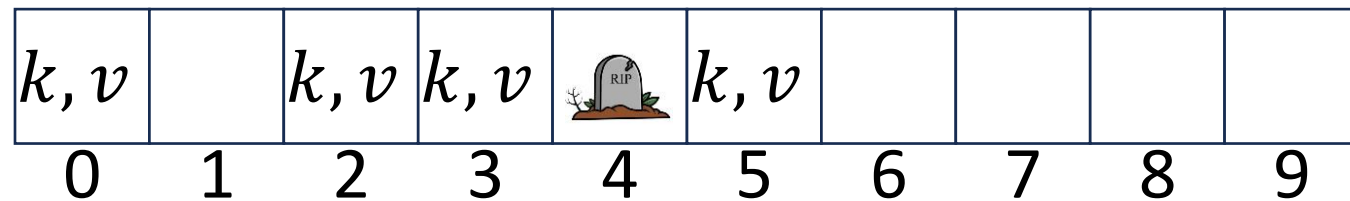
Linear Probing: Delete

- To delete key k , where $h(k) \% \text{table.length} = i$
 - Assume it is present
- Beginning at index i , probe until we find k (call this location index j)
- Mark j as empty (e.g. null), then...
 - Challenge: we need to make sure future finds could be successful
 - What if there were values that mapped to index i that appeared after j ?
 - What if there were items that hashed to a value between i and j and appeared after j due to probing?



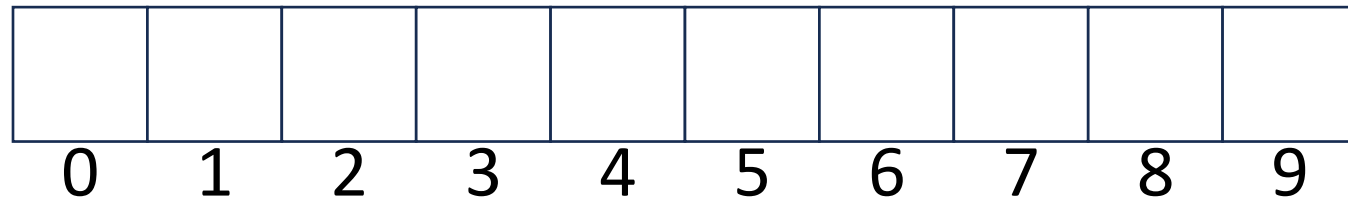
Linear Probing: Delete

- **Option 1 (harder):** Plug the hole with other items in a way that makes probes behave correctly
- **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



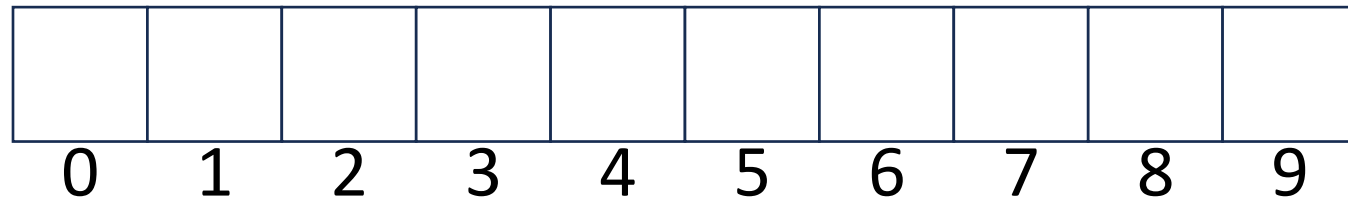
Linear Probing + Tombstone: Find

- To find key k
 - Calculate $i = h(k) \% \text{table.length}$
 - While `table[i]` has a key other than k , set $i = (i+1) \% \text{table.length}$
 - If you come across k return `table[i]`
 - If you come across an empty index, the find was unsuccessful



Linear Probing + Tombstone: Insert

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

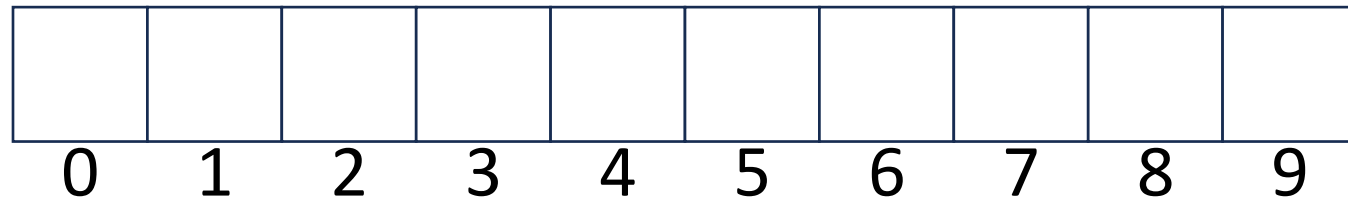


Downsides of Linear Probing

- 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
 - Feedback loop!
- What happens when λ exceeds 1?
 - Impossible!
 - You can't insert more stuff

Quadratic Probing: Insert Procedure

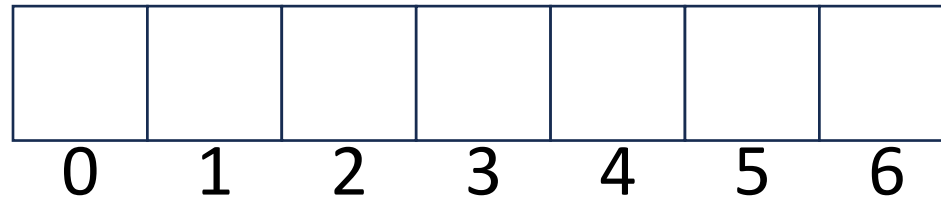
- To insert k, v
 - Calculate $i = h(k) \% \text{table.length}$
 - If $\text{table}[i]$ is occupied then try $(i+1^2) \% \text{table.length}$
 - If that is occupied try $(i+2^2) \% \text{table.length}$
 - If that is occupied try $(i+3^2) \% \text{table.length}$
 - If that is occupied try $(i+4^2) \% \text{table.length}$
 - ...



Quadratic Probing: Example

- Insert:

- 76
- 40
- 48
- 5
- 55
- 47



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 hash to the same value

Double Hashing: Insert Procedure

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

