

CSE 332 Summer 2026

Lecture 8: AVL Wrapup, Hashing

Michael Whitmeyer

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

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	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$

AVL Trees

- 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:
 - Do a BST delete, then rotate if the tree is unbalanced (apply LL, RR, LR, RL cases as needed from leaf to root)

Other Tree-based Dictionaries

- 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

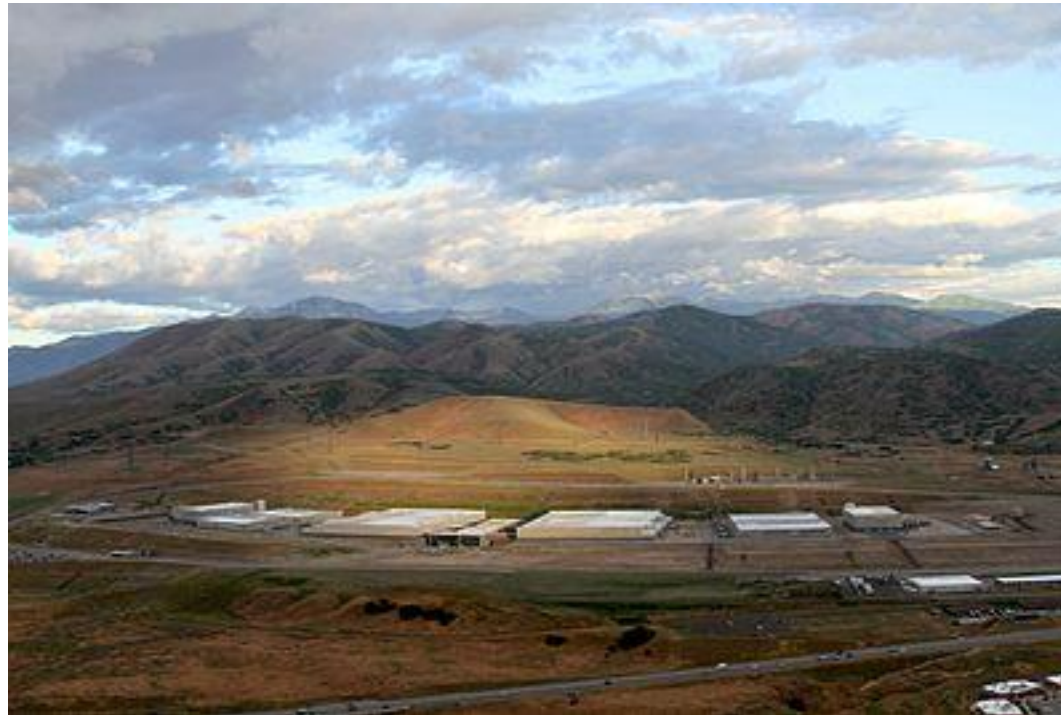
- Contents:
 - Sets of key+value pairs
 - ~~Keys must be comparable~~ Keys have a hash function
- Operations:
 - insert(key, value)
 - Adds the (key,value) pair into the dictionary
 - If the key already has a value, overwrite the old value
 - Consequence: Keys cannot be repeated
 - find(key)
 - Returns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

The Best Dictionary Data Structure!

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

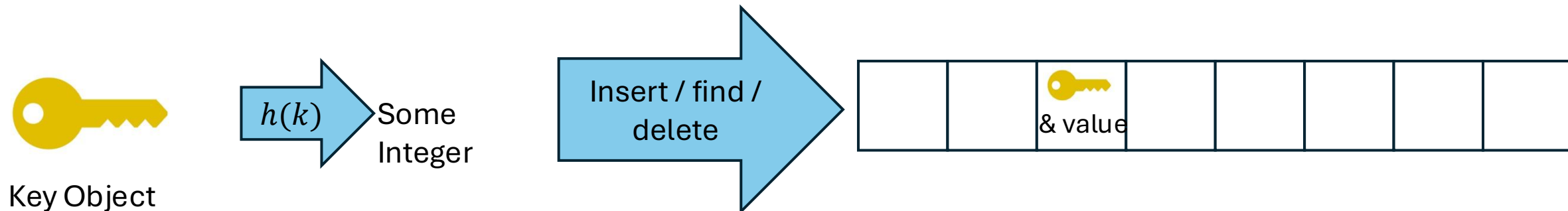
```
insert(key, value){  
    arr[key]=value;  
}  
find(key){  
    return arr[key];  
}  
delete(key){  
    arr[key] = null;  
}
```

Problem?



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



Example

0	1	2	3	4	5	6	7	8	9

- Key: Phone Number
- Value: People
- Table size: 10
- $h(phone) = \text{number as an integer}$
- $h(8675309) \% 10 = 9$

What Influences Running time?

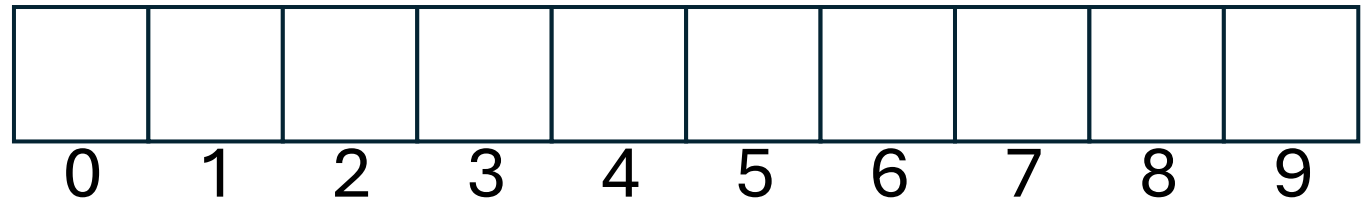
- How long hashing itself takes
- Likelihood of collisions
 - Size of the array vs number of values in the array
 - “quality” of our hash function
- What we do when we have a collision

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 on 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

A Bad Hash (and phone number trivia)

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



Compare These Hash Functions (for strings)

- Let $s = s_0s_1s_2 \dots s_{m-1}$ be a string of length m
 - Let $a(s_i)$ be the ascii encoding of the character s_i
- $h_1(s) = a(s_0)$
- $h_2(s) = \left(\sum_{i=0}^{m-1} a(s_i)\right)$
- $h_3(s) = \left(\sum_{i=0}^{m-1} a(s_i) \cdot 37^i\right)$
- $h_4(s) = \left(2 \cdot \sum_{i=0}^{m-1} a(s_i) \cdot 37^i\right)$

Properties of Those Example Hash Functions

- Let $s = s_0s_1s_2 \dots s_{m-1}$ 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
- $h_3(s) = \left(\sum_{i=0}^{m-1} a(s_i) \cdot 37^i\right)$
 - Is: Consistent, efficient, uniform, and effective
- $h_4(s) = \left(2 \cdot \sum_{i=0}^{m-1} a(s_i) \cdot 37^i\right)$
 - Is: Consistent, efficient, effective

Ideal Insert procedure

Supposing we have a “good” hash function:

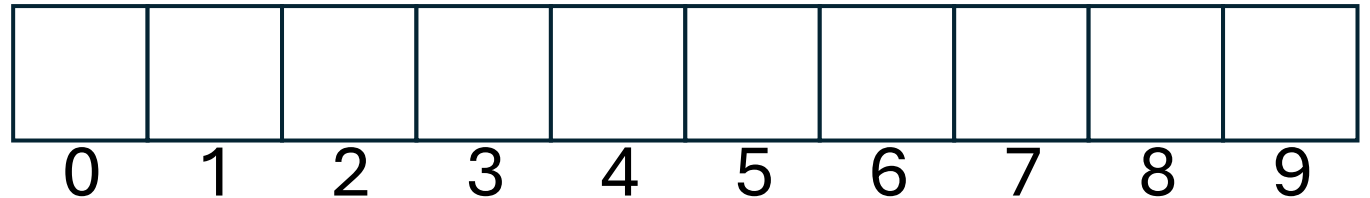
```
insert(key, value){  
    h = key.hash();  
    arr[h % table.length] = value;  
}
```

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

This is called a “collision”

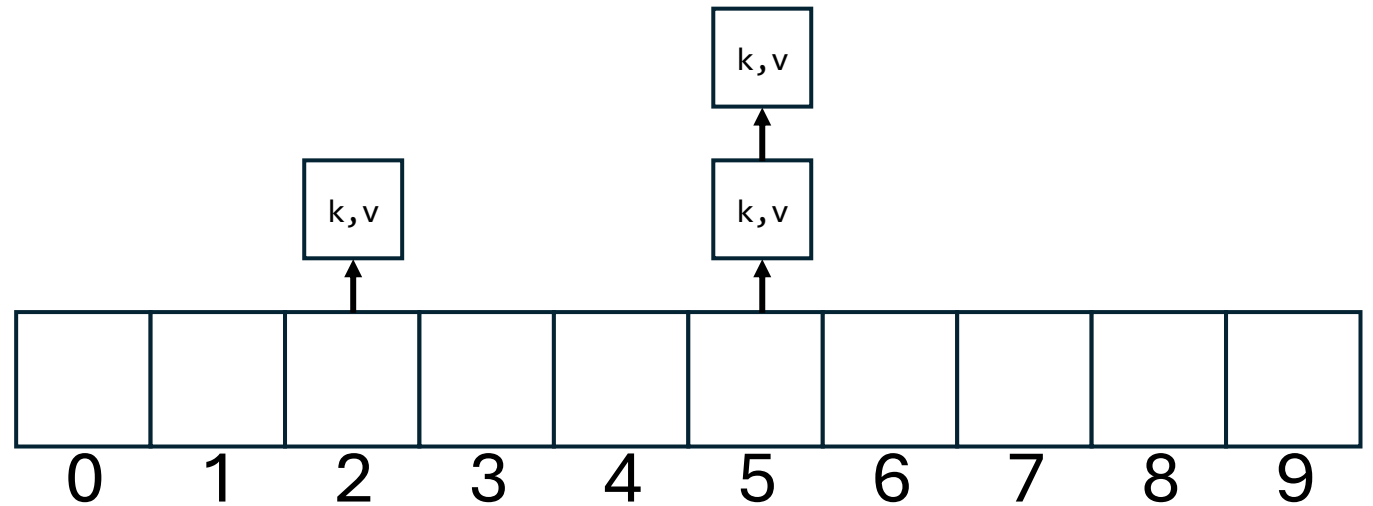
Collision Resolution

- 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



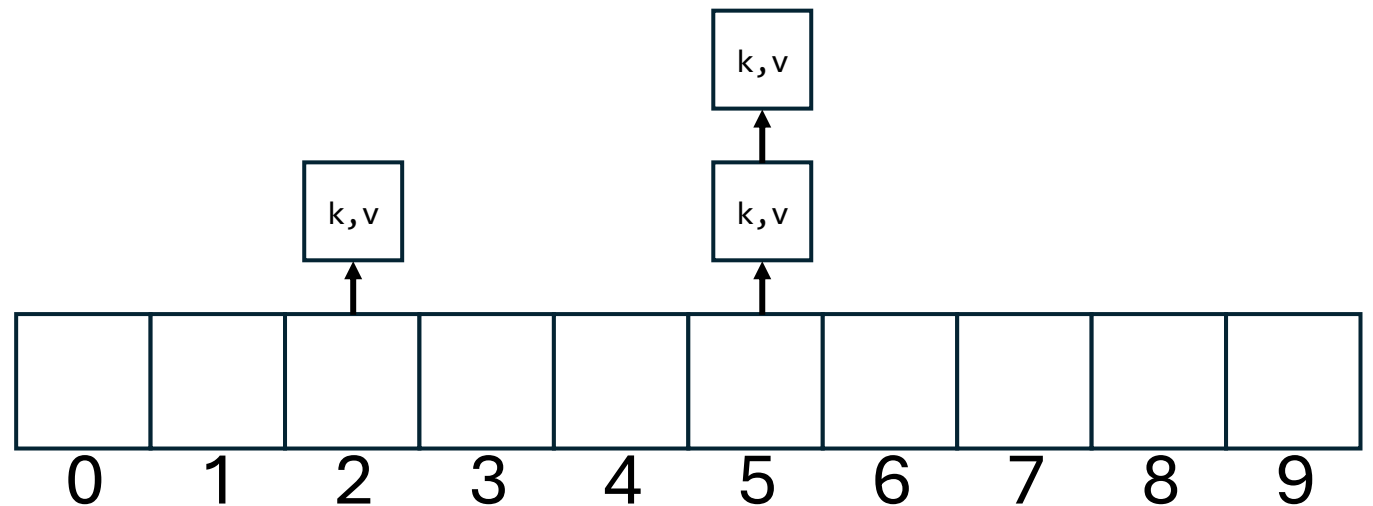
Separate Chaining Insert

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



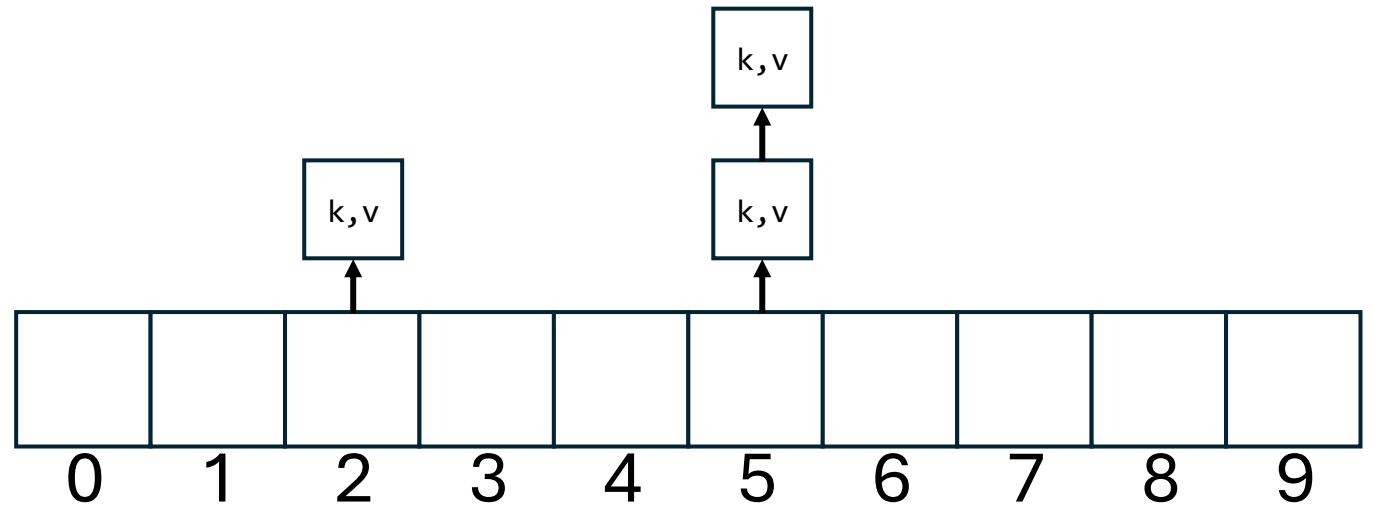
Separate Chaining Find

- 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 $\text{table}[i]$



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