CSE 332 Autumn 2023
Lecture 1~~2~~: Hashing

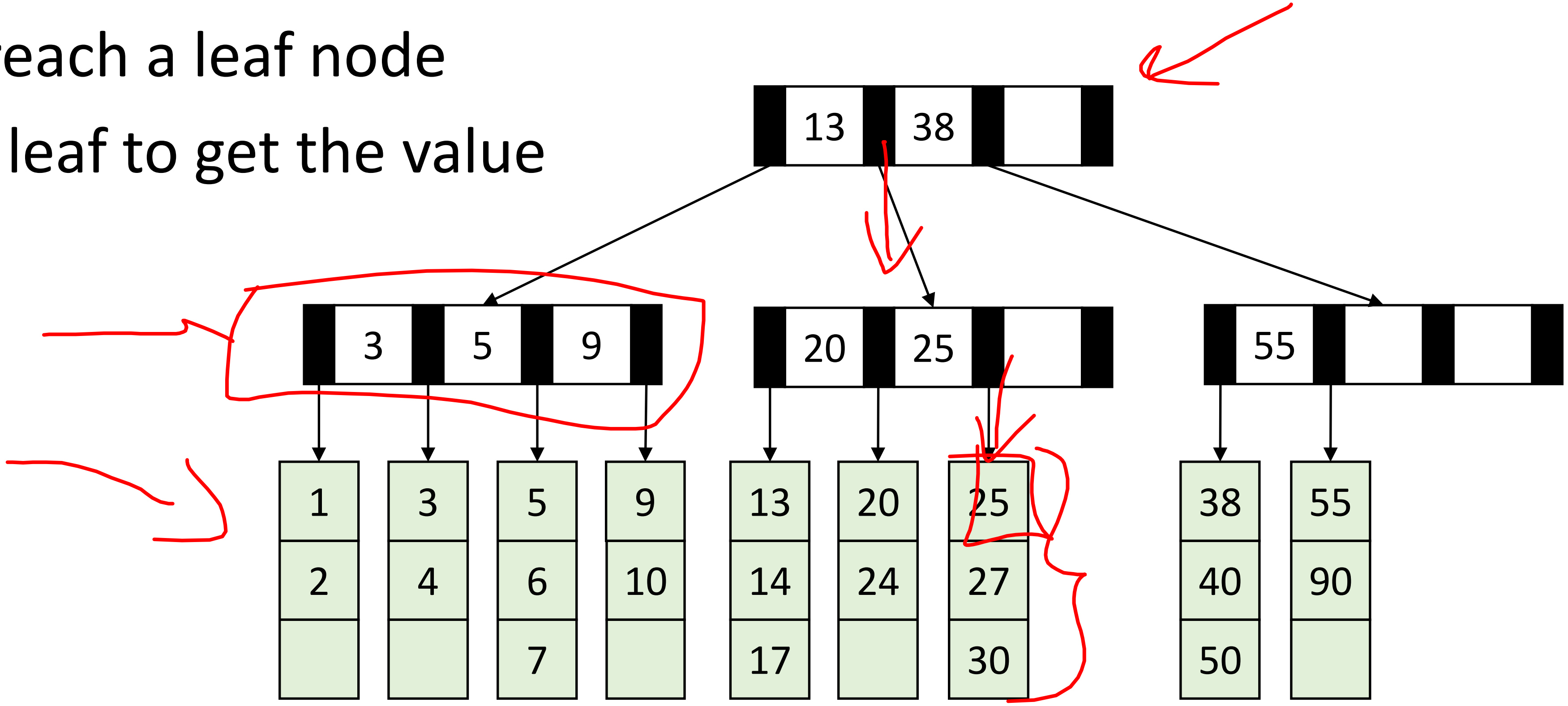
Nathan Brunelle

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

Find

- Start at the root node
- Binary search internal nodes to identify correct subtree
- Repeat until you reach a leaf node
- Binary search the leaf to get the value

25



Insertion Summary

- Binary search to find which leaf should contain the new item
- If there's room, add it to the leaf array (maintaining sorted order)
- If there's not room, **split**
  - Make a new leaf node, move the larger $\left\lfloor \frac{L+1}{2} \right\rfloor$ items to it
  - If there's room in the parent internal node, add new leaf to it (with new key bound value)
  - If there's not room in the parent internal node, **split** that!
    - Make a new internal node and have it point to the larger $\left\lfloor \frac{M+1}{2} \right\rfloor$
    - If there's room in the parent internal node, add this internal node to it
    - If there's not room, repeat this process until there is!

Insertion TLDR

- Find where the item goes by repeated binary search
- If there's room, just add it
- If there's not room, split things until there is

Running Time of Find

M, L

$2\left\lceil \frac{n}{L} \right\rceil$

- Maximum number of leaves:
  - $\frac{2n}{L}$
  - $\boxed{\Theta\left(\frac{n}{L}\right)}$
- Maximum height of the tree:
  - $2 \log_M \frac{2n}{L}$
  - $\boxed{\Theta\left(\log_M \frac{n}{L}\right)}$
- Find:
  - One binary search per level of the tree
    - $\Theta(\log_2 M)$ per search
  - One binary search in the leaf
    - $\Theta(\log_2 L)$

Overall: $\Theta\left(\log_2 M \cdot \log_M \frac{n}{L} + \log_2 L\right)$

Usually simplified to:

$\Theta(\log_2 M \cdot \log_M n)$

$\log_{M/2}\left(\frac{n}{L}\right)$

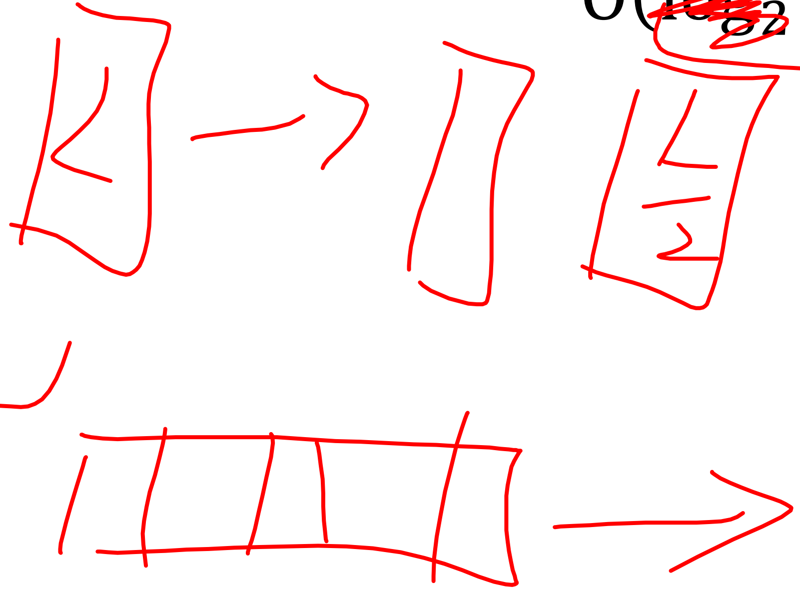
Running Time of Insert

- Find:
  - $\Theta(\log_2 M \cdot \log_M n)$
- Add item to leaf:
  - $\Theta(L)$
- Split a leaf
  - $\Theta(L)$
- Split one internal node:
  - $\Theta(M)$

Overall: $\Theta(L + M \cdot \log_M n)$

Usually simplified to:

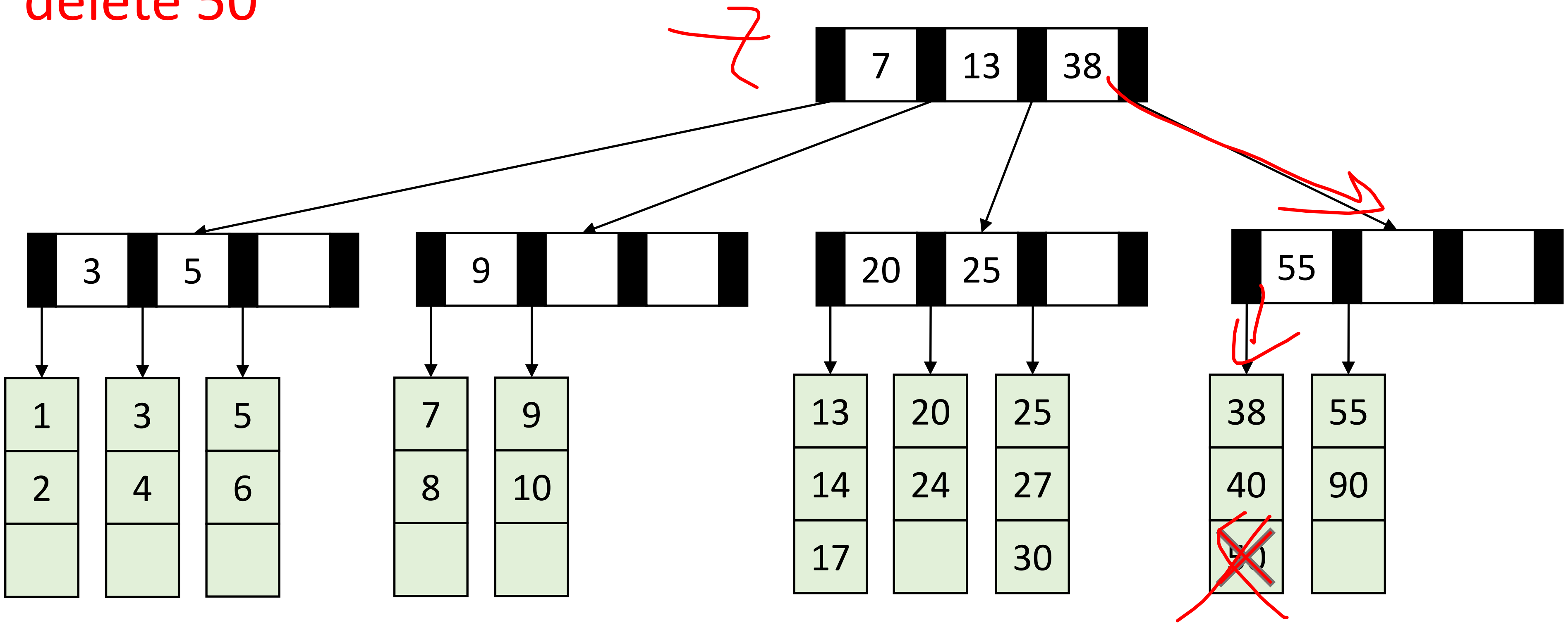
$$\Theta(\cancel{\log_2} M \cdot \log_M n)$$



Delete

- Recall: all nodes must be at least half full (except root at startup)

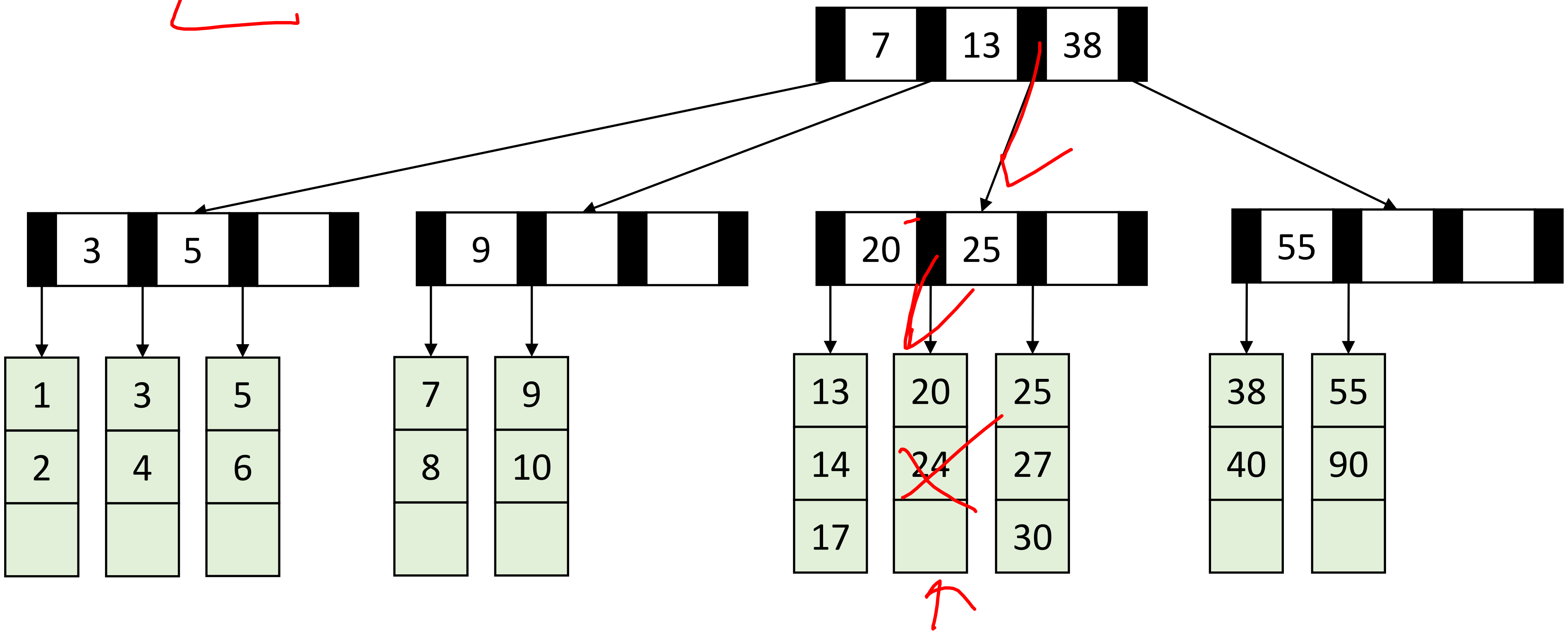
delete 50



Delete

- Recall: all nodes must be at least half full (except root at startup)

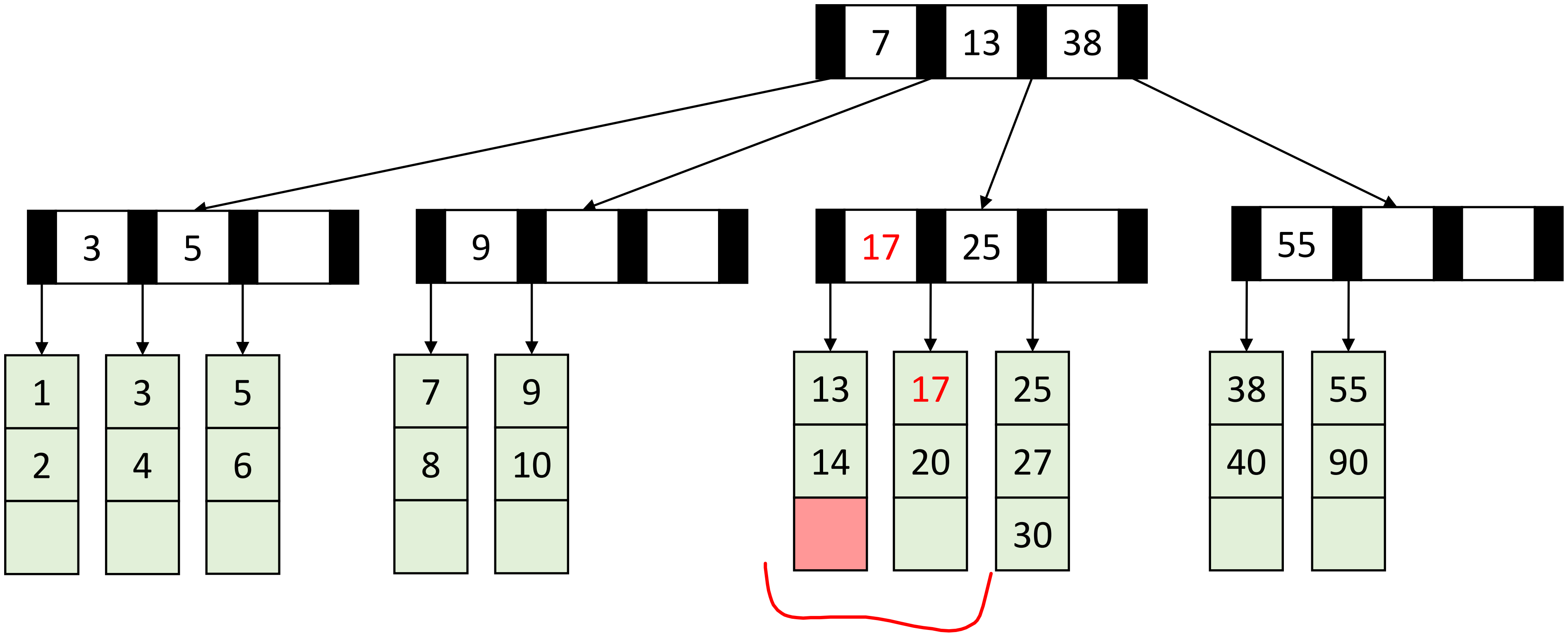
delete 24



Delete

- Recall: all nodes must be at least half full (except root at startup)

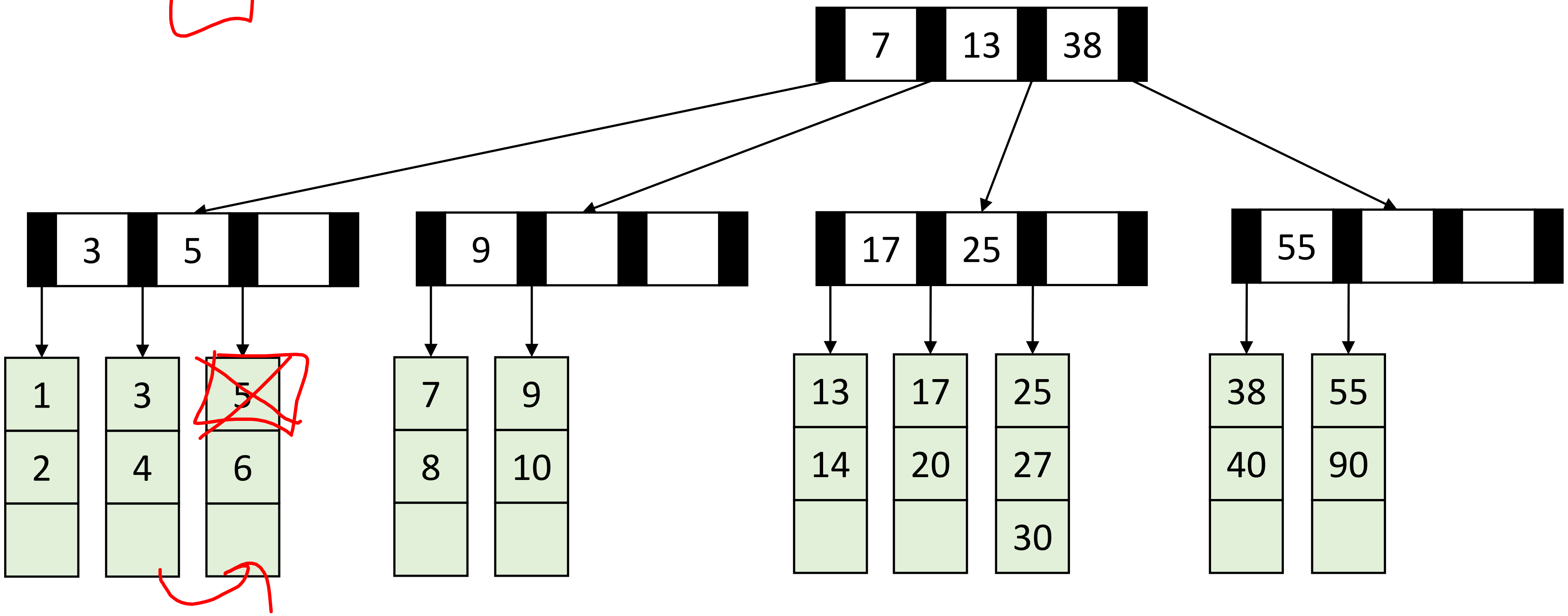
delete 24



Delete

- Recall: all nodes must be at least half full (except root at startup)

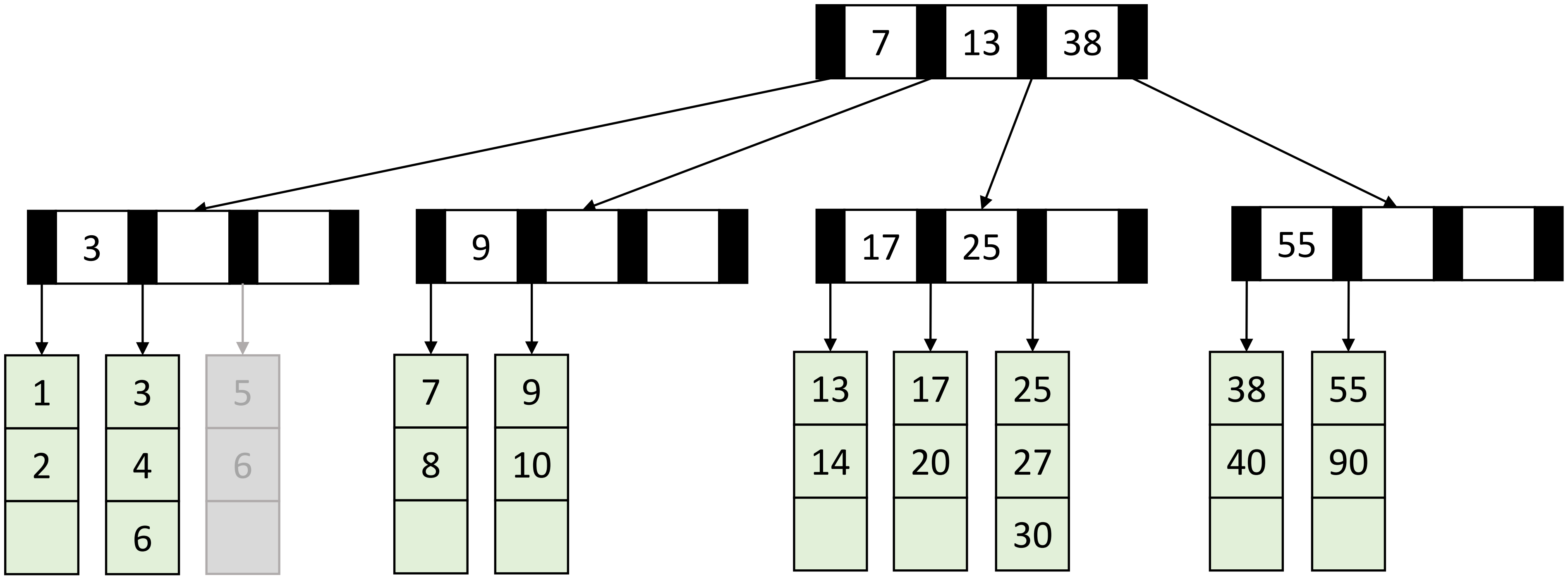
delete 5



Delete

- Recall: all nodes must be at least half full (except root at startup)

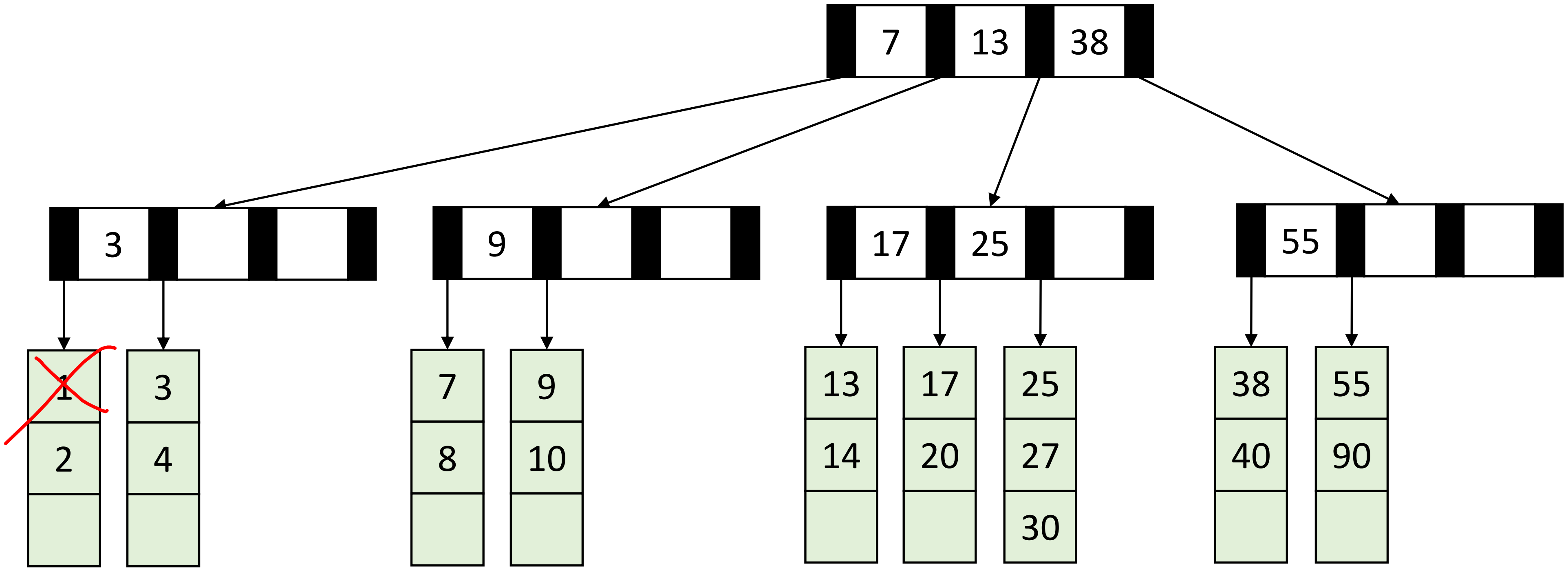
delete 5



Delete

- Recall: all nodes must be at least half full (except root at startup)

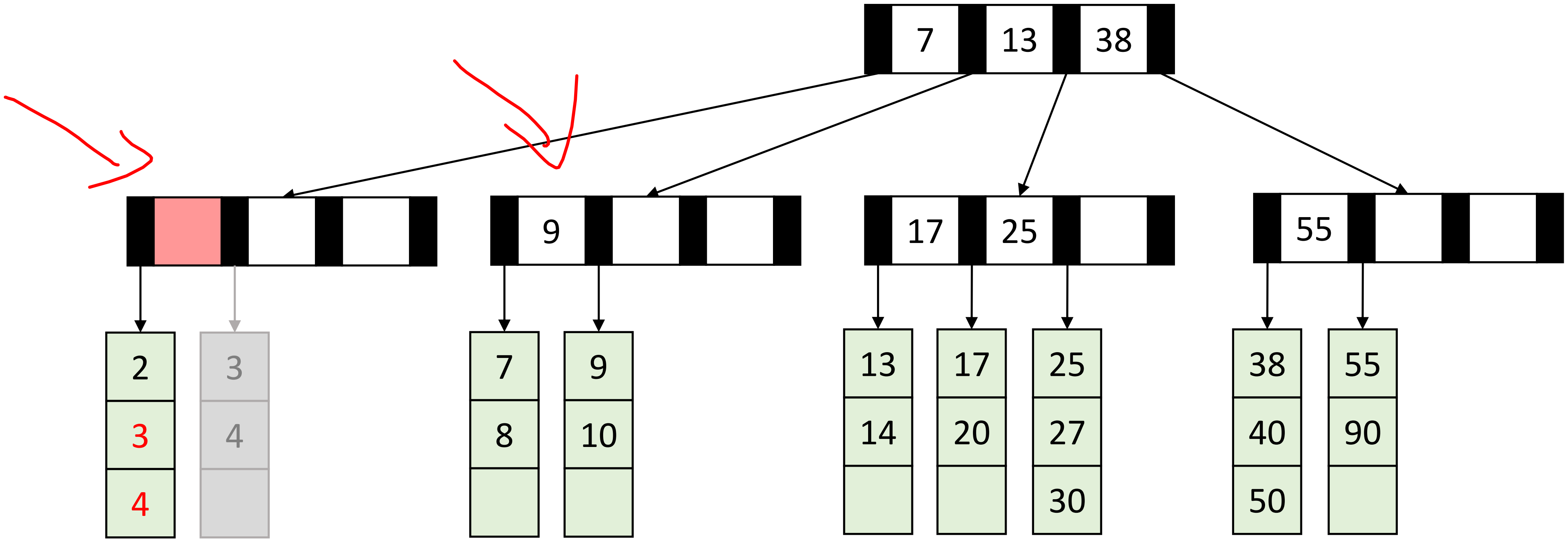
delete 1



Delete

- Recall: all nodes must be at least half full (except root at startup)

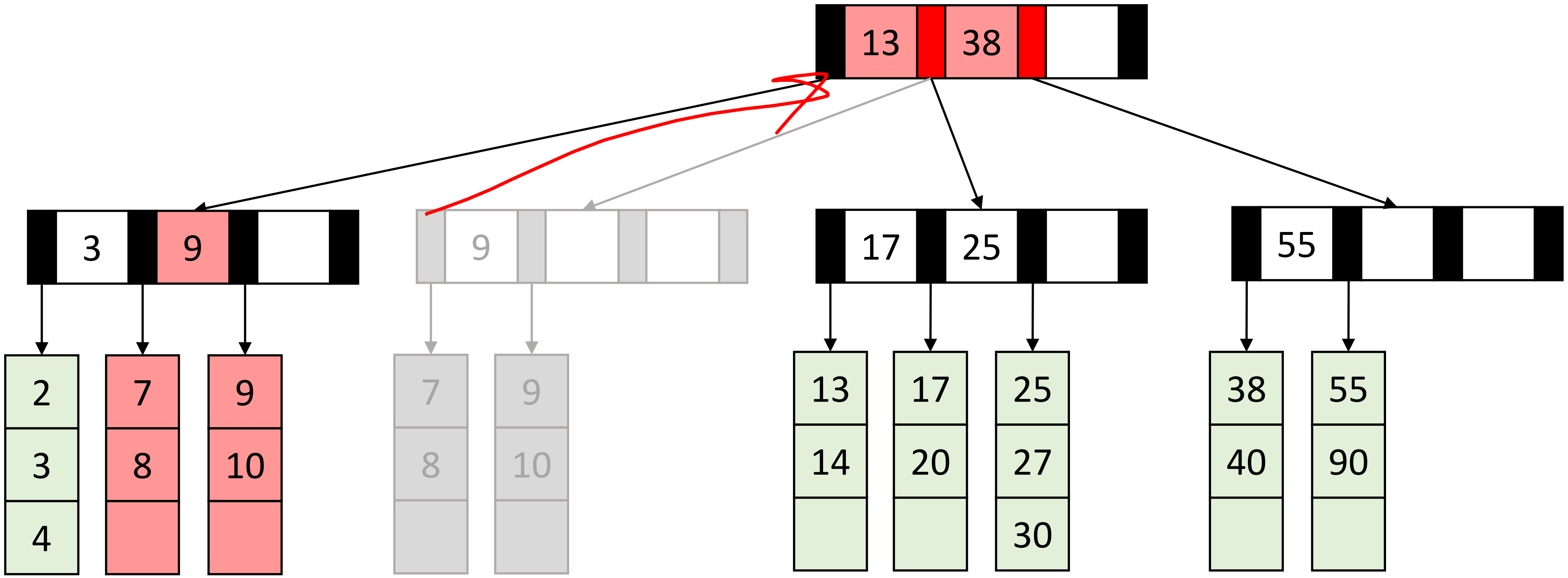
delete 1



Delete

- Recall: all nodes must be at least half full (except root at startup)

delete 1



Delete Summary

- Find the item
- Remove the item from the leaf
    - If that causes the leaf to be under-full, adopt from a neighbor
    - If that would cause the neighbor to be under-full, merge them
    - Update the parent
        - If that causes the parent to be under-full, adopt from a neighbor
        - If that causes the neighbor to be under-full, merge
        - Update the parent
            - …

Delete TLDR

- Find and remove from leaf
- Keep doing this until everything is “full enough”:
  - If the node is now too small, adopt from a neighbor
  - If the neighbor is too small then merge

Next topic: Hash Tables

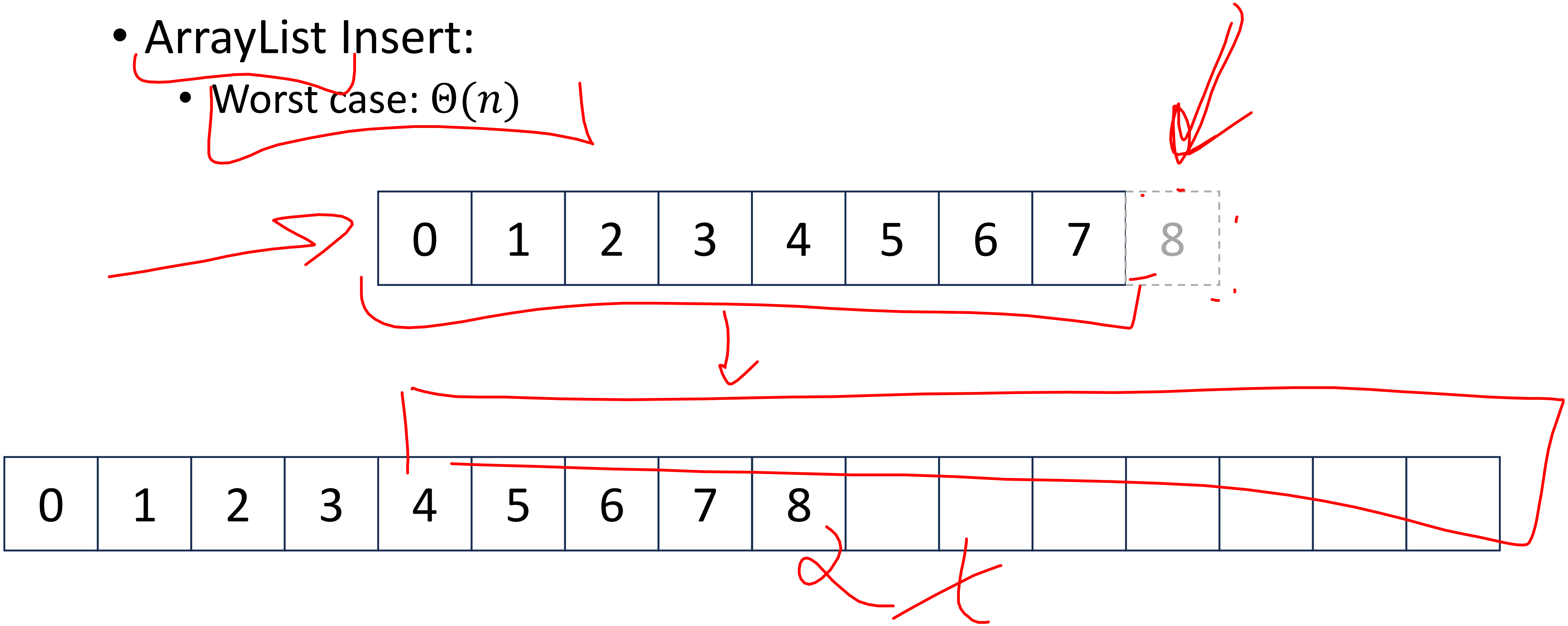
Data Structure	Time to insert	Time to find	Time to delete
Unsorted Array	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Unsorted Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Sorted Array	$\Theta(n)$	$\Theta(\log n)$	$\Theta(n)$
Sorted Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$
Binary Search Tree	$\Theta(n)$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

Two Different ideas of "Average"

- Expected Time
  - The expected number of operations a randomly-chosen input uses
  - Assumed randomness from somewhere
    - Most simply: from the input
    - Preferably: from the algorithm/data structure itself
  - $f(n) =$ sum of the running times for each input of size n divided by the number of inputs of size n
- Amortized Time ← ArrayList<
  - The long-term average per-execution cost (in the worst case)
  - Rather than look at the worst case of one execution, look at the total worst case of a sequential chain of many executions
    - Why? The worst case may be guaranteed to be rare
  - $f(n) =$ the sum of the running times from a sequence of n sequential calls to the function divided by n

Amortized Example

- ArrayList Insert:
  - Worst case: $\Theta(n)$



Amortized Example

- ArrayList Insert:
    - First 8 inserts: 1 operation each
    - 9th insert: 9 operations
    - Next 7 inserts: 1 operation each
    - 17th insert: 17 operations
    - Next 15 inserts: 1 operation each
    - …

Do x operations with cost 1
Do 1 operation with cost x
Do x operations with cost 1
Do 1 operation with cost $\boxed{2x}$
Do $2x$ operations with cost 1
Do 1 operation with cost $4x$
Do $4x$ operations with cost 1
Do 1 operation with cost $8x$
…
Amortized: each operation cost 2 operations
$\Theta(1)$

$2x+1$

$2x+$

2^+

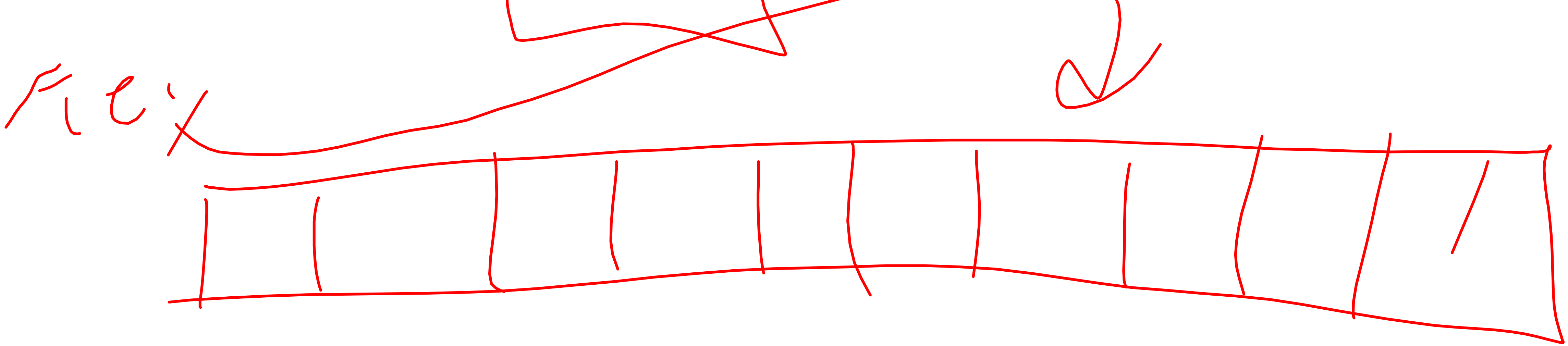
2

0	1	2	3	4	5	6	7	8

0	1	2	3	4	5	6	7	8							

Hash Tables

- Motivation:
  - Why not just have a gigantic array?

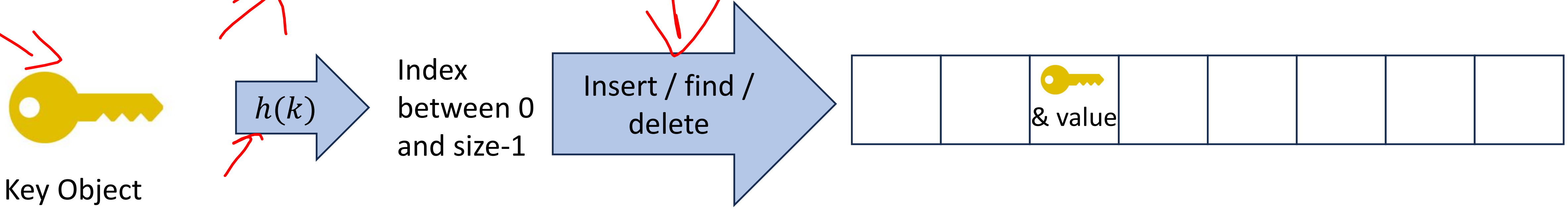


Problem? ~8 exabytes

6

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
- $h(phone)$ = number as an integer % 10
- $h(8675309) = 9$

What Influences Running time?

- Rarity of collisions -
  - quality of hash
- Size of the array

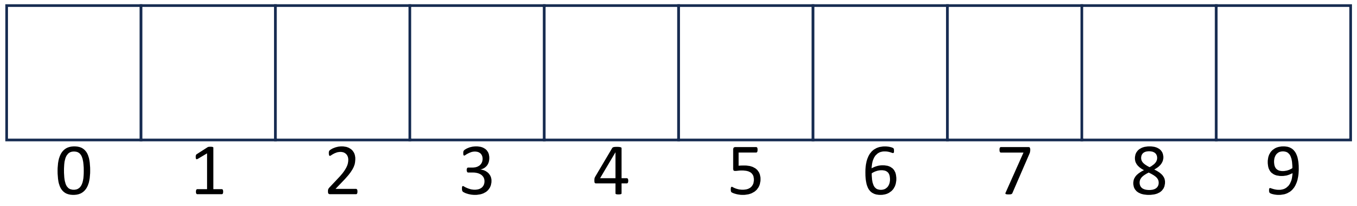
Properties of a “Good” Hash

- Definition: A hash function maps objects to integers

- Should be very efficient
  - Calculating the hash should be negligible
- Should “randomly” scatter objects
  - Even similar objects should be able to be far away
- Should use the entire table
  - There should not be any indices in the table that nothing can hash to
  - Picking a table size that is prime helps with this
- Should use things needed to “identify” the object
  - Use only fields you would check for a .equals method be included in calculating the hash
  - More fields typically leads to fewer collisions, but less efficient calculation

A Bad Hash (and phone number trivia)

- $h(phone) =$ the first digit of the phone number
  - No US phone numbers start with 1 or 0
  - If we're sampling from this class, 2 is by far the most likely

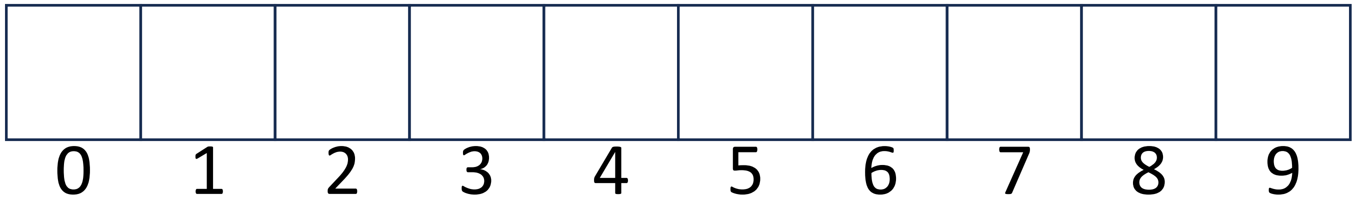


Compare These Hash Functions (for strings)

- Let $s = s_0 s_1 s_2 \dots s_{m-1}$ be a string of length m
  - Let $a(s_i)$ be the ascii encoding of the character s_i
- $h_1(s) = a(s_0)$
- $h_2(s) = \left(\sum_{i=0}^{m-1} a(s_i)\right)$
- $h_3(s) = \left(\sum_{i=0}^{m-1} a(s_i) \cdot 37^i\right)$

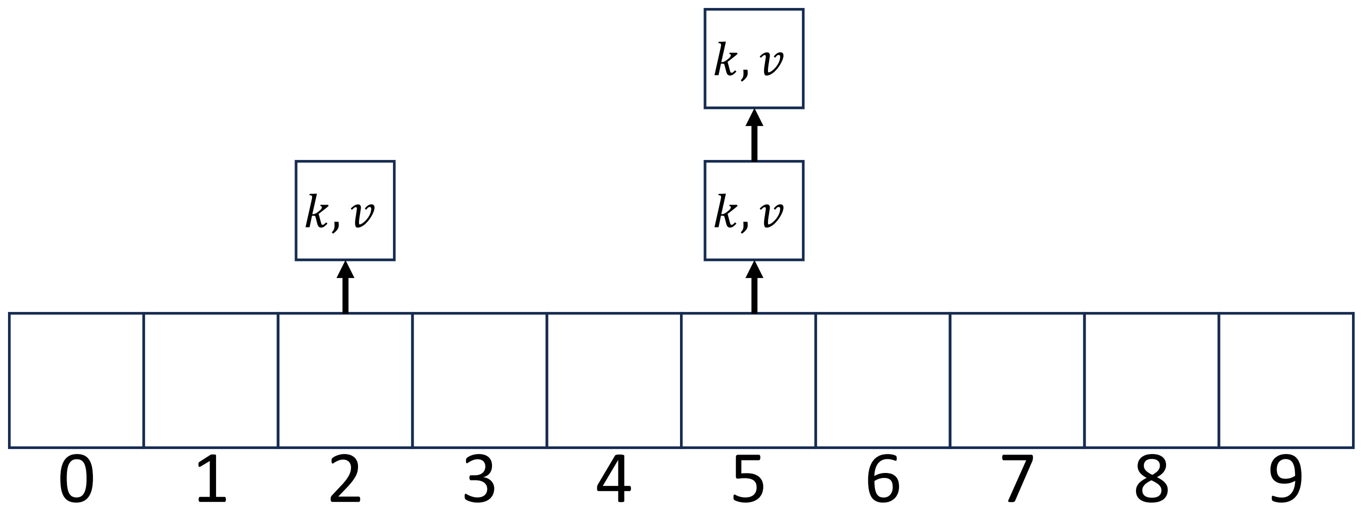
Collision Resolution

- 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



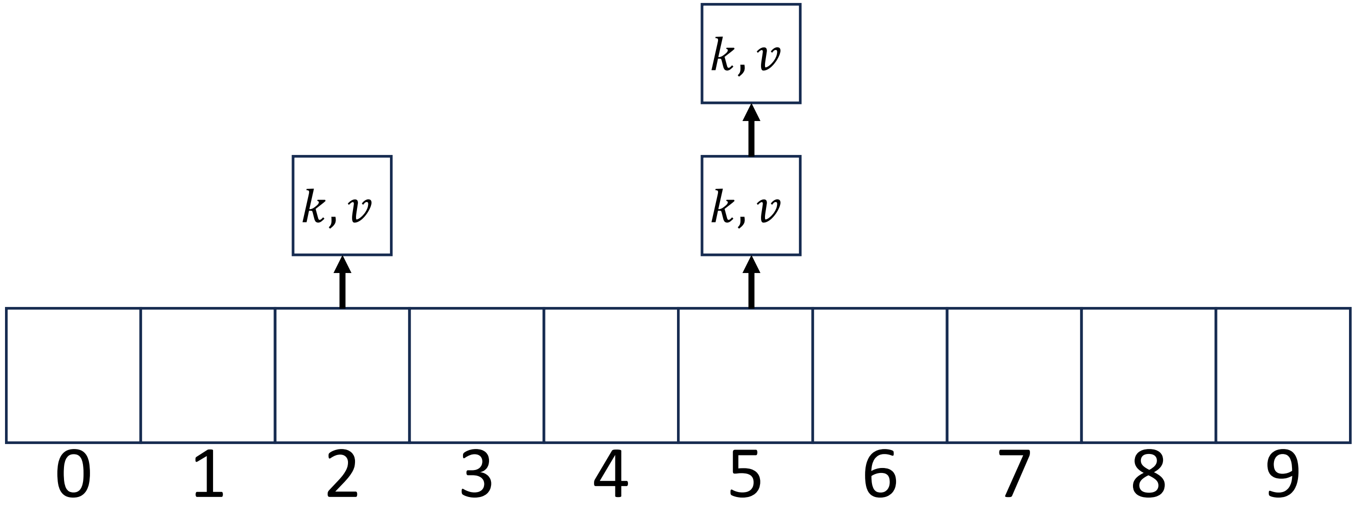
Separate Chaining Insert

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



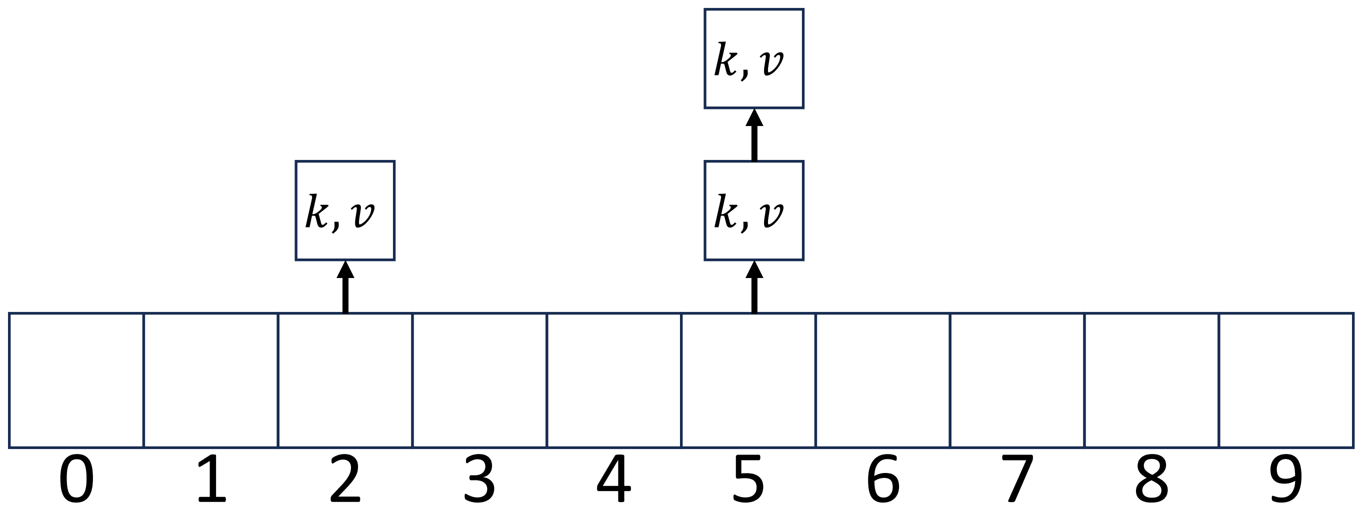
Separate Chaining Find

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



Separate Chaining Delete

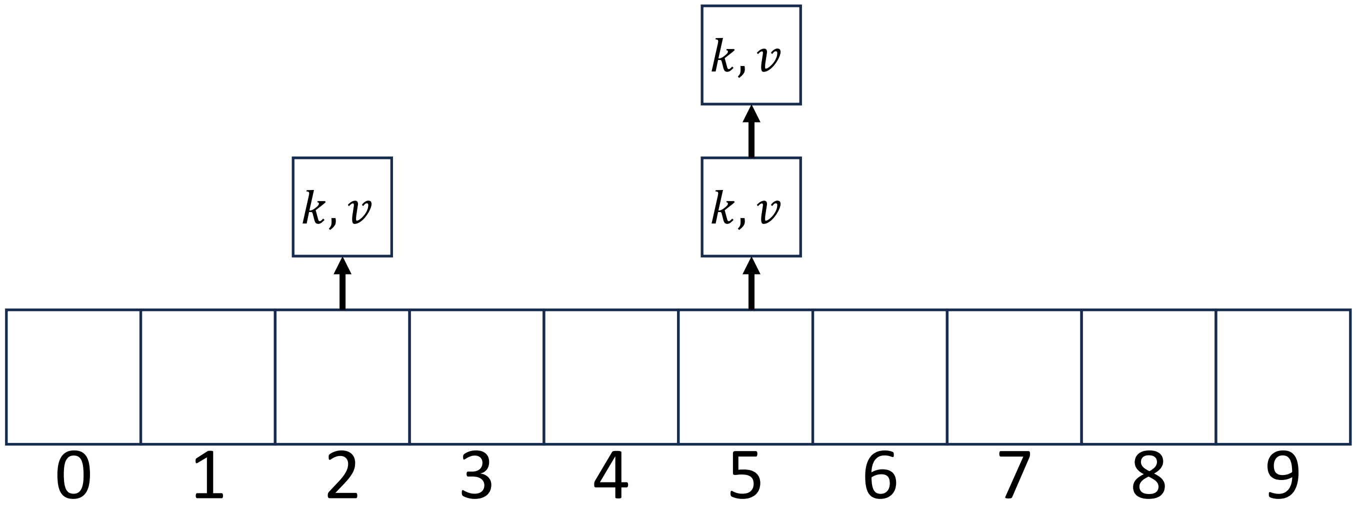
- To delete k:
  - Compute the index using $i = h(k) \ \% \ size$
  - Call delete with the key on the data structure at $table[i]$



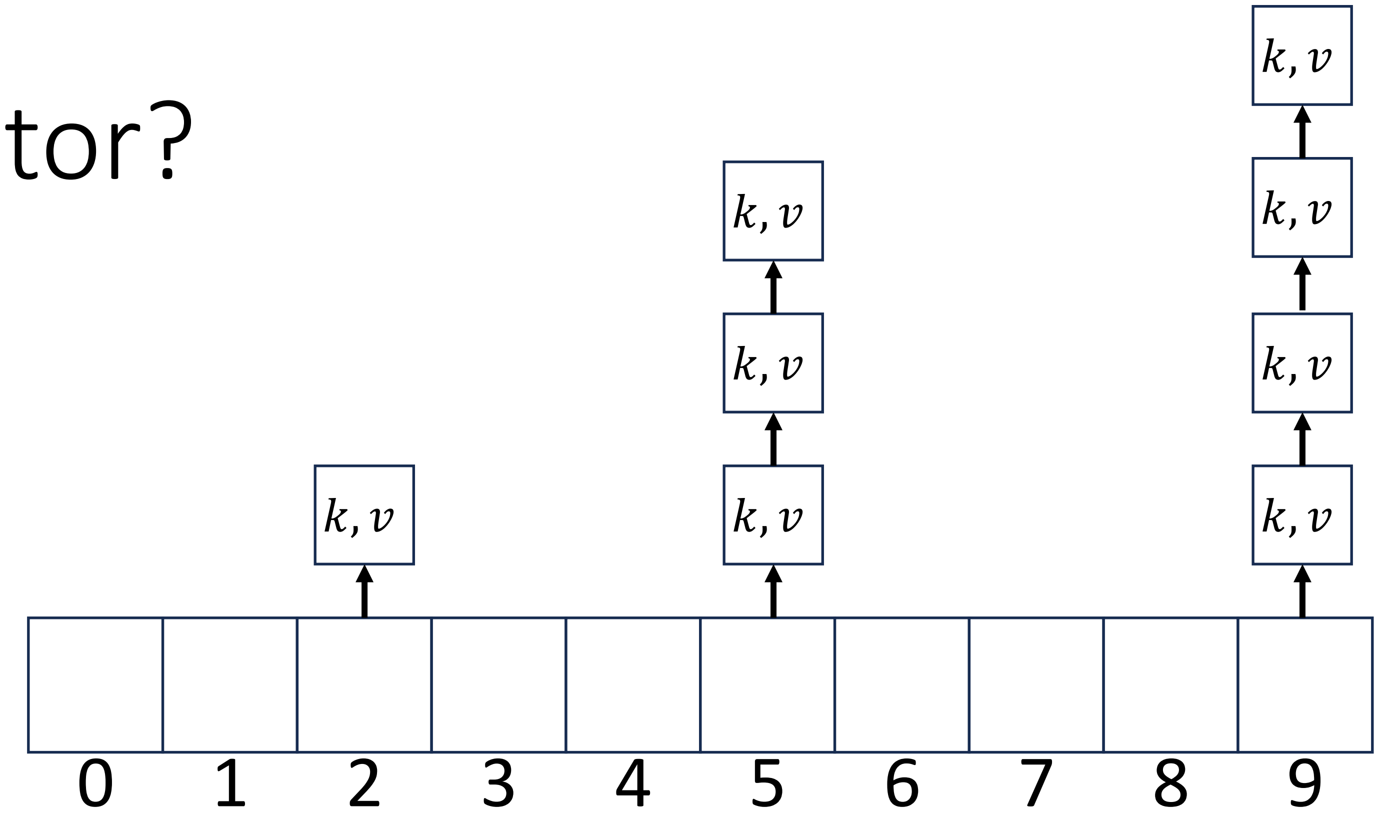
Formal Running Time Analysis

- The **load factor** of a hash table represents the average number of items per “bucket”
  - $\lambda = \frac{n}{size}$
- Assume we have a has 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 $\Theta(1)$?

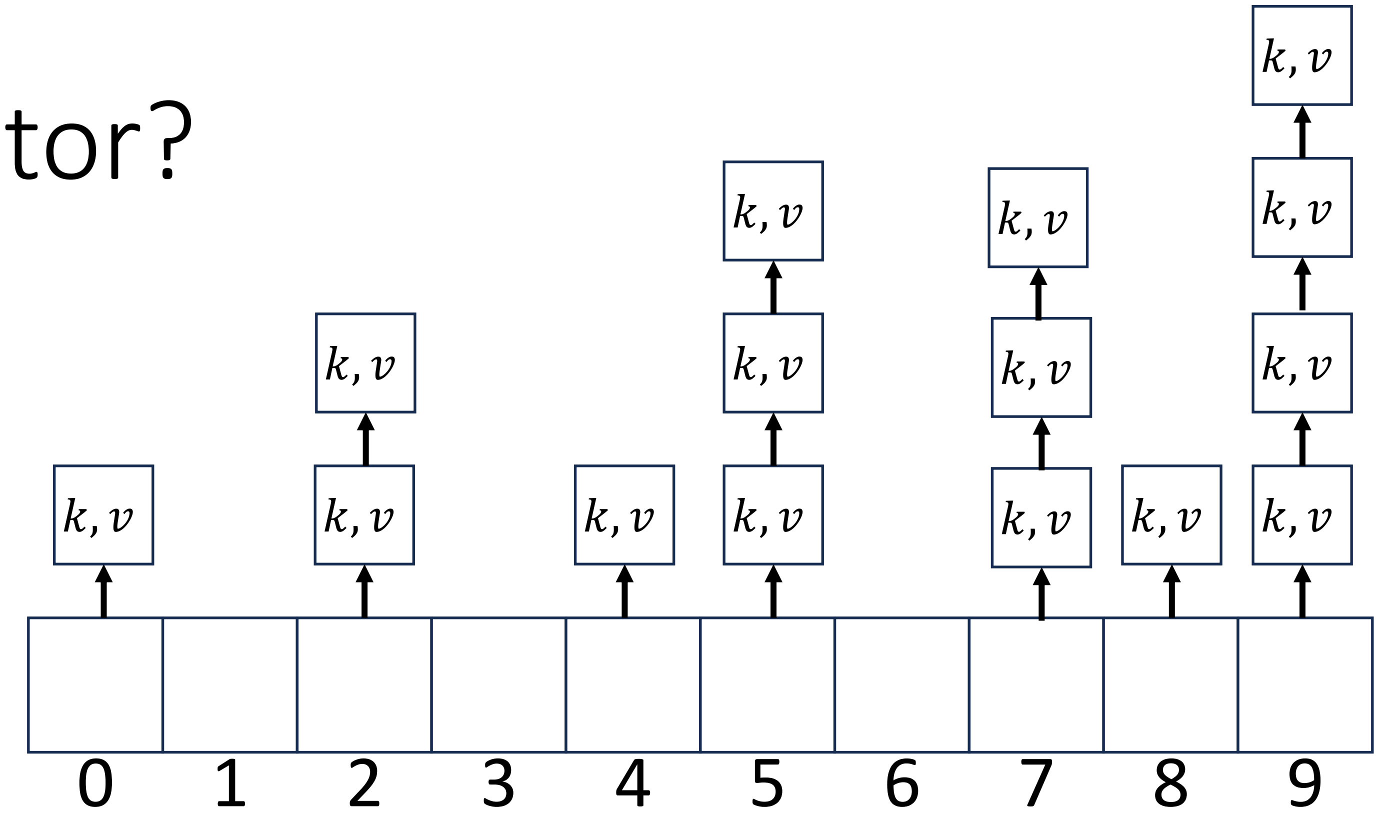
Load Factor?



Load Factor?



Load Factor?



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 $i = h(k)\ \%\ size$
  - If $table[i]$ is occupied then try $(i+1)\%\ size$
  - If that is occupied try $(i+2)\%\ size$
  - If that is occupied try $(i+3)\%\ size$
  - …

0	1	2	3	4	5	6	7	8	9

Linear Probing: Find

- Let’s do this together!

Linear Probing: Find

- To find key k
  - Calculate $i = h(k)\ \%\ size$
  - If $table[i]$ is occupied and does not contain k then look at $(i + 1)\ \%\ size$
  - If that is occupied and does not contain k then look at $(i + 2)\ \%\ size$
  - If that is occupied and does not contain k then look at $(i + 3)\ \%\ size$
  - Repeat until you either find k or else you reach an empty cell in the table

Linear Probing: Delete

- Let’s do this together!

Linear Probing: Delete

- Let’s do this together!