# CSE100: Design and Analysis of Algorithms Lecture 15 – Binary Search Trees (wrap up) and Hashing

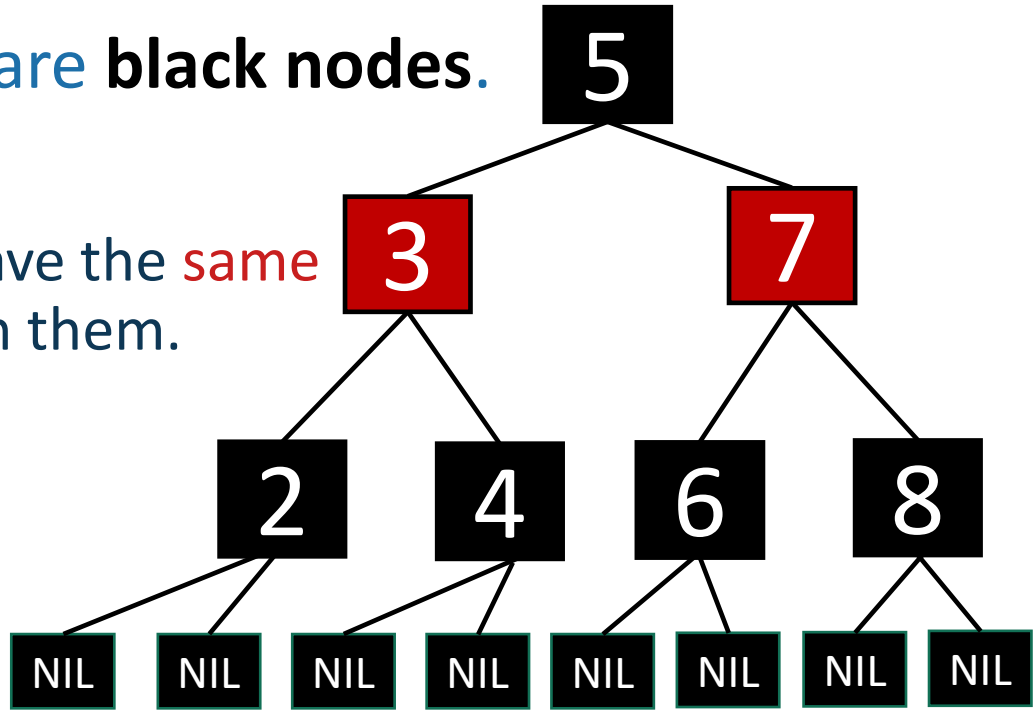
**Mar 10th 2022**

Red-Black Trees and Hashing

# Red-Black Trees (review)

obey the following rules (which are a proxy for balance)

1. Every node is colored **red** or **black.**
2. The root node is a **black node**.
3. NIL children count as **black nodes**.
4. Children of a **red node** are **black nodes**.
5. For all nodes x:
   - all paths from x to NIL's have the same number of **black nodes** on them.




I'm not going to draw the NIL children in the future, but they are treated as black nodes.

# The height of a RB-tree with n non-NIL nodes is at most $2\log(n+1)$ (review)

- Define bh(x) to be the number of black nodes in any path from x to NIL.
  - (excluding x, including NIL).
- Claim:
  - There are at least $2^{bh(x)} - 1$ non-NIL nodes in the subtree underneath x. (Including x).




Claim: at least $2^{b(x)} - 1$ nodes in this WHOLE subtree (of any color).

Then:

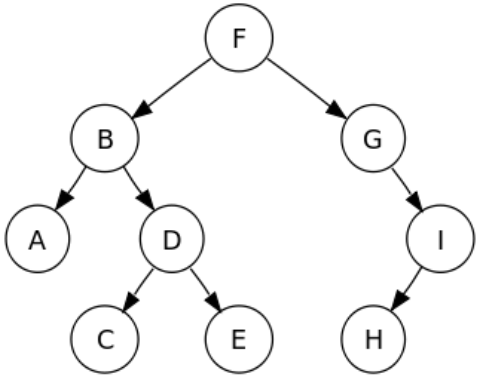
$n \geq 2^{bh(root)} - 1$ using the Claim

$\geq 2^{height/2} - 1$ bh(root) >= height/2 because of RBTree rules.

Rearranging:

$$n + 1 \geq 2^{height/2} \Rightarrow height \leq 2\log(n+1)$$

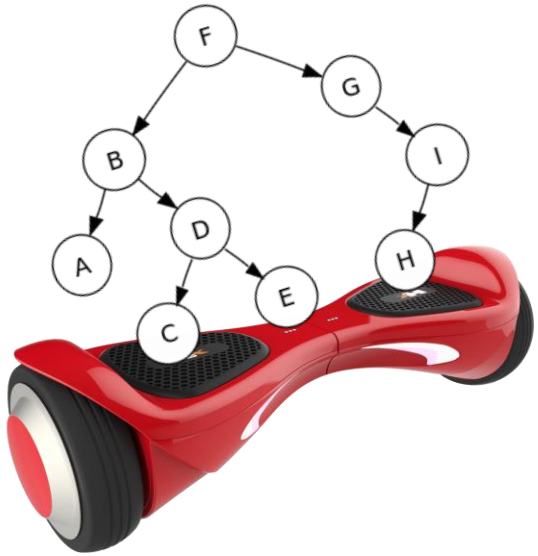
# Today (part 1)




- Begin a brief foray into data structures!
- Binary search trees
  - You may remember these from CSE 30
  - They are better when they're balanced.

this will lead us to…

- Self-Balancing Binary Search Trees
  - **Red-Black** trees. (warp up)




# This is great!

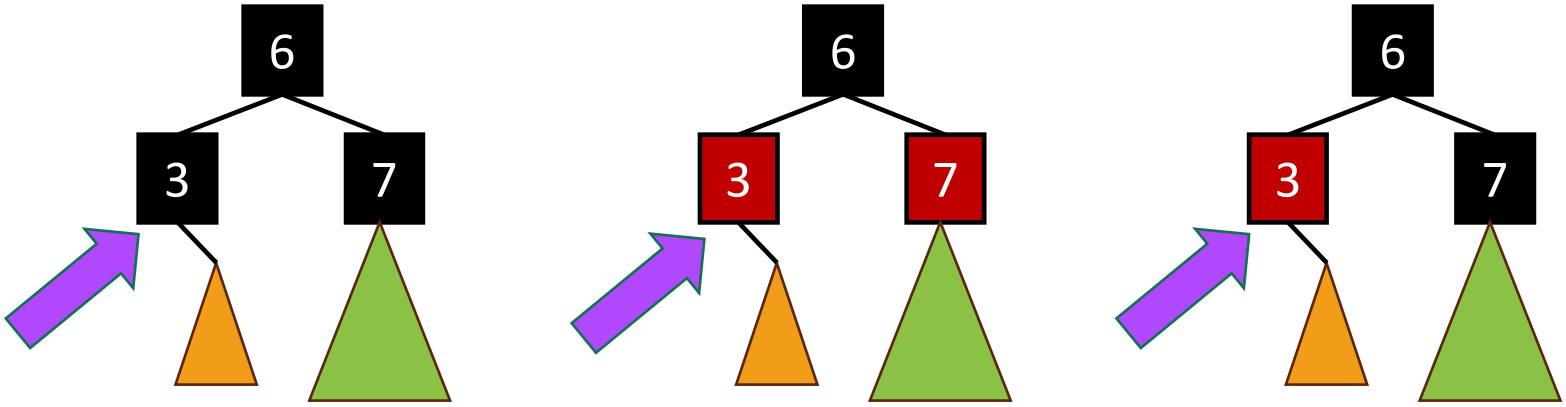
- SEARCH in an RBTree is immediately O(log(n)), since the depth of an RBTree is O(log(n)).

- What about INSERT/DELETE?
  - Turns out, you can INSERT and DELETE items from an RBTree in time O(log(n)), while maintaining the RBTree property.

# INSERT/DELETE

- For the rest of lecture [if time], we'll sketch how to do INSERT/DELETE for RBTrees.
  - See CLRS for more details if you are interested.
- You are **not responsible** for the details of INSERT/DELETE for RBTrees for this class.
  - You should know what the "proxy for balance" property is and why it ensures approximate balance.
  - You should know **that** this property can be efficiently maintained, but you do not need to know the details of how.

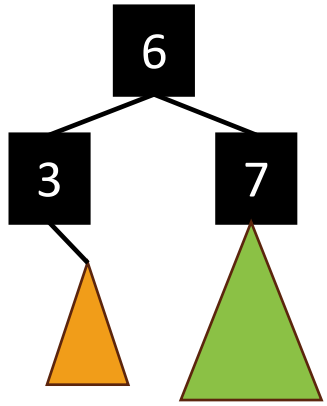
# INSERT: Many cases



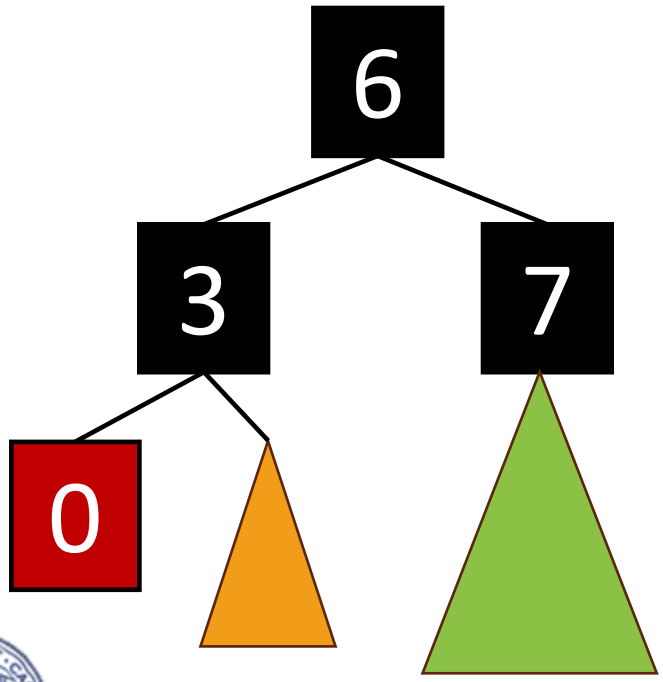

- Suppose we want to insert 0 **here**.
- There are 3 “important” cases for different colorings of the existing tree, and there are 9 more cases for all of the various symmetries of these 3 cases.

# INSERT: Case 1

- Make a new **red node**.
- Insert it as you would normally.

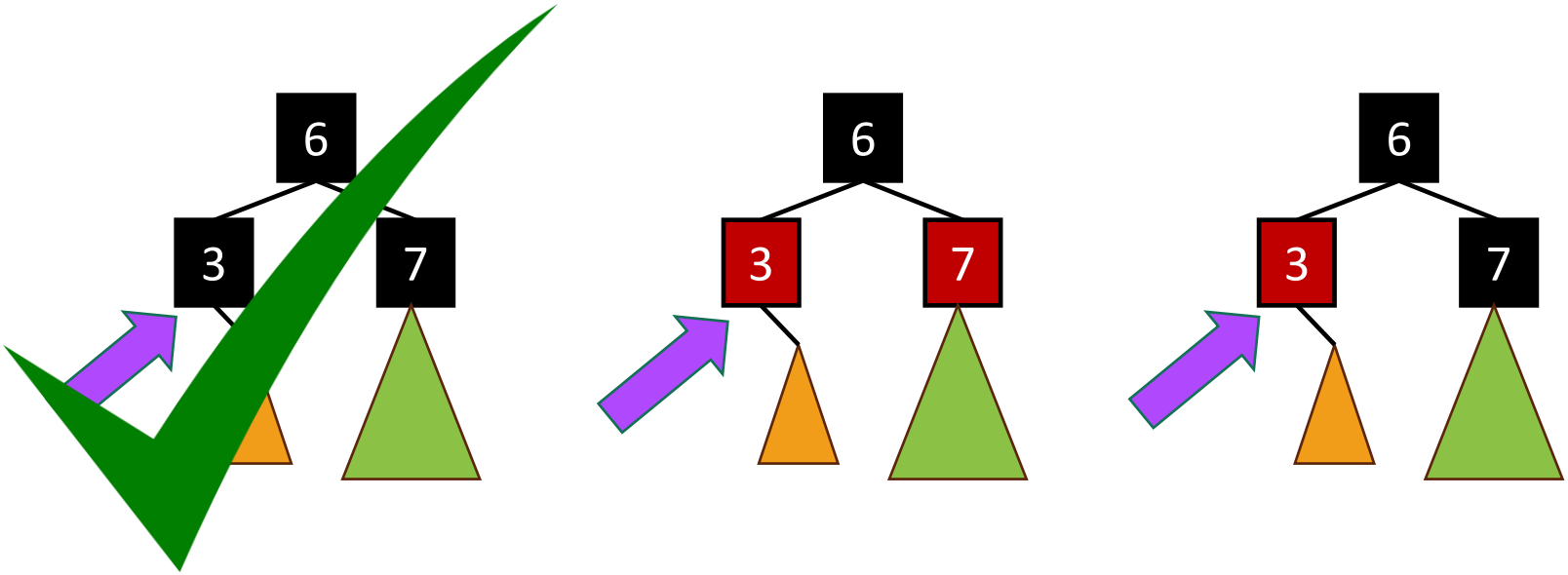



What if it looks like this?




Example: insert 0

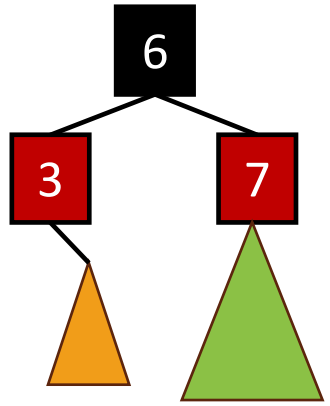
# INSERT: Many cases




- Suppose we want to insert 0 **here**.
- There are 3 “important” cases for different colorings of the existing tree, and there are 9 more cases for all of the various symmetries of these 3 cases.

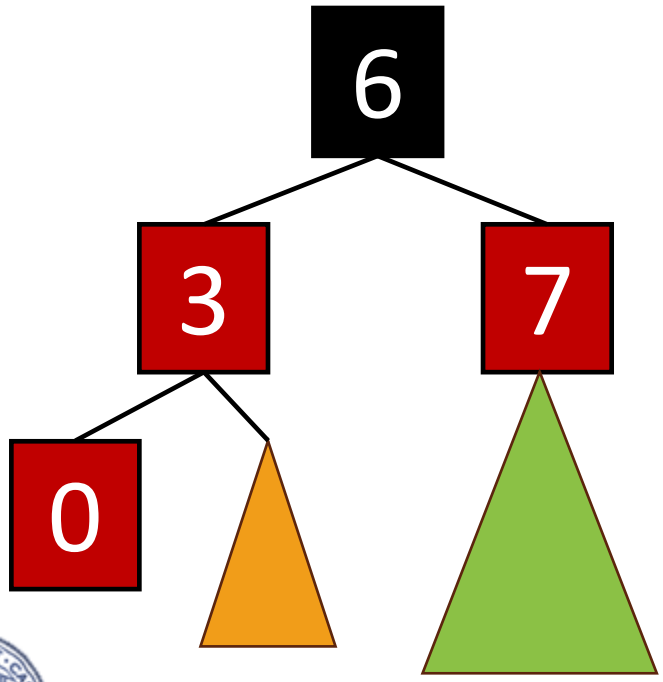
# INSERT: Case 2

- Make a new **red node**.
- Insert it as you would normally.
- Fix things up if needed.



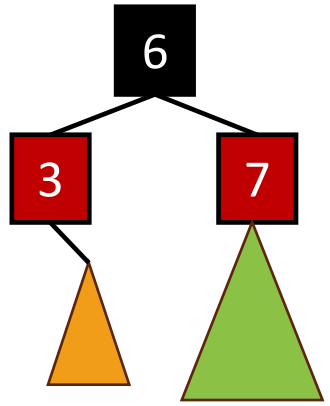

What if it looks like this?

Example: insert 0

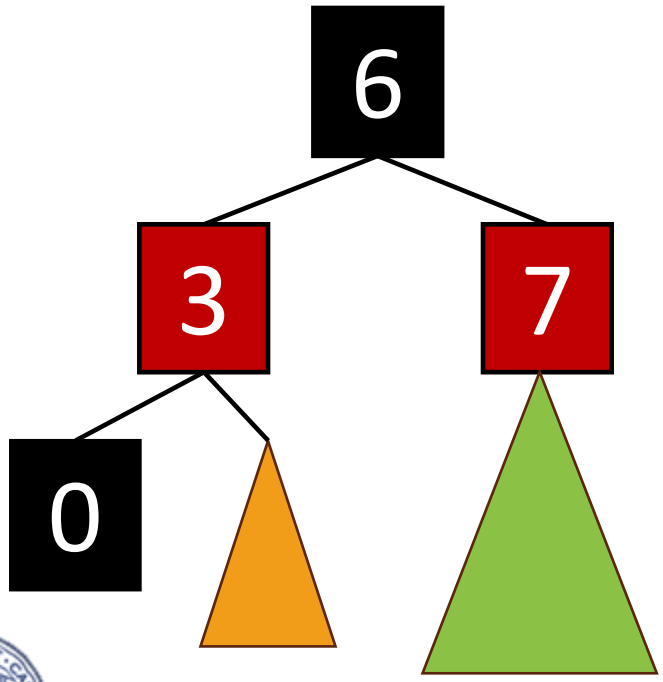



# INSERT: Case 2

- Make a new **red node**.
- Insert it as you would normally.
- Fix things up if needed.



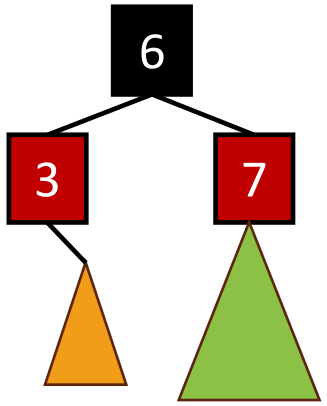

What if it looks like this?




Example: insert 0

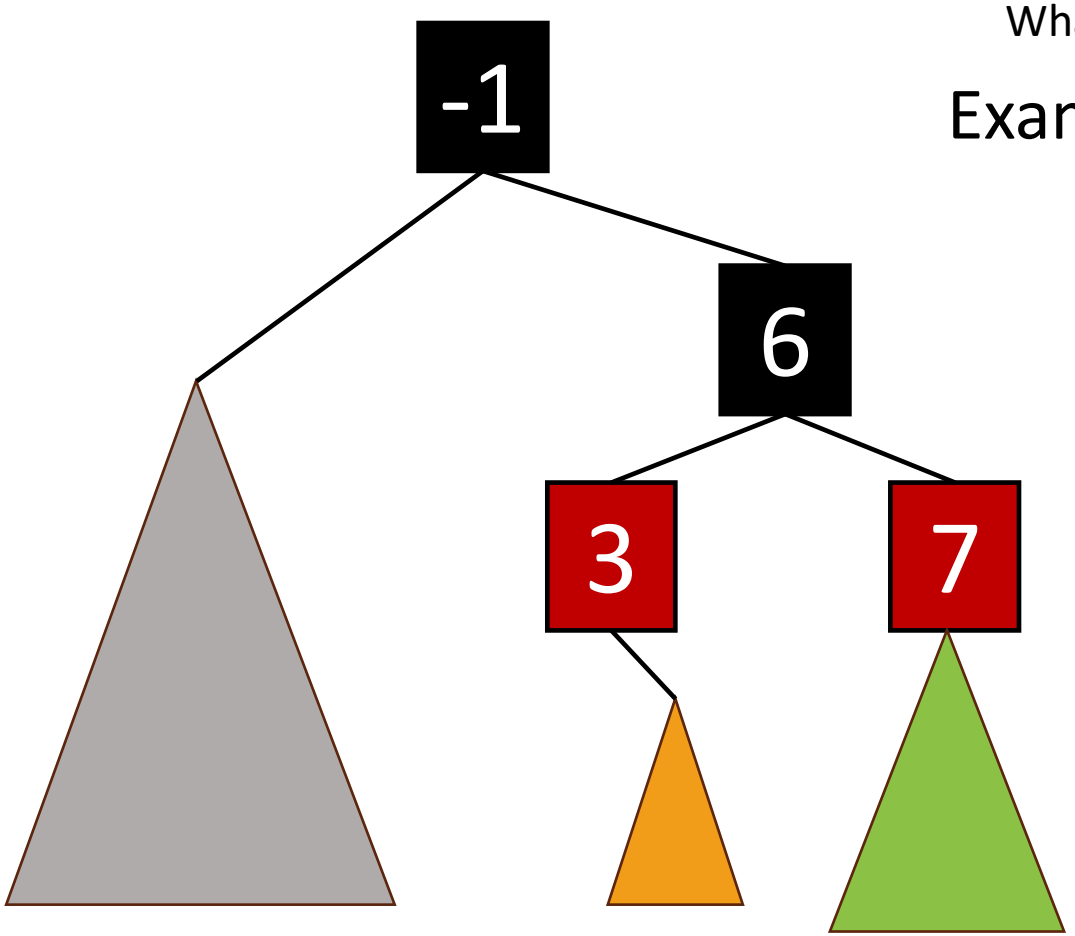
Can't we just insert 0 as a **black node?**


# We need a bit more context



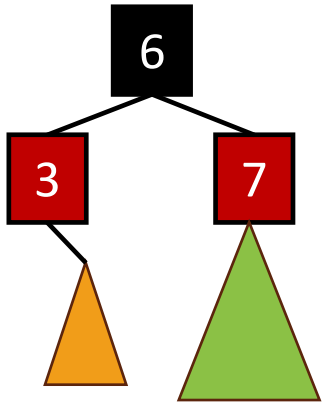

What if it looks like this?

Example: insert 0



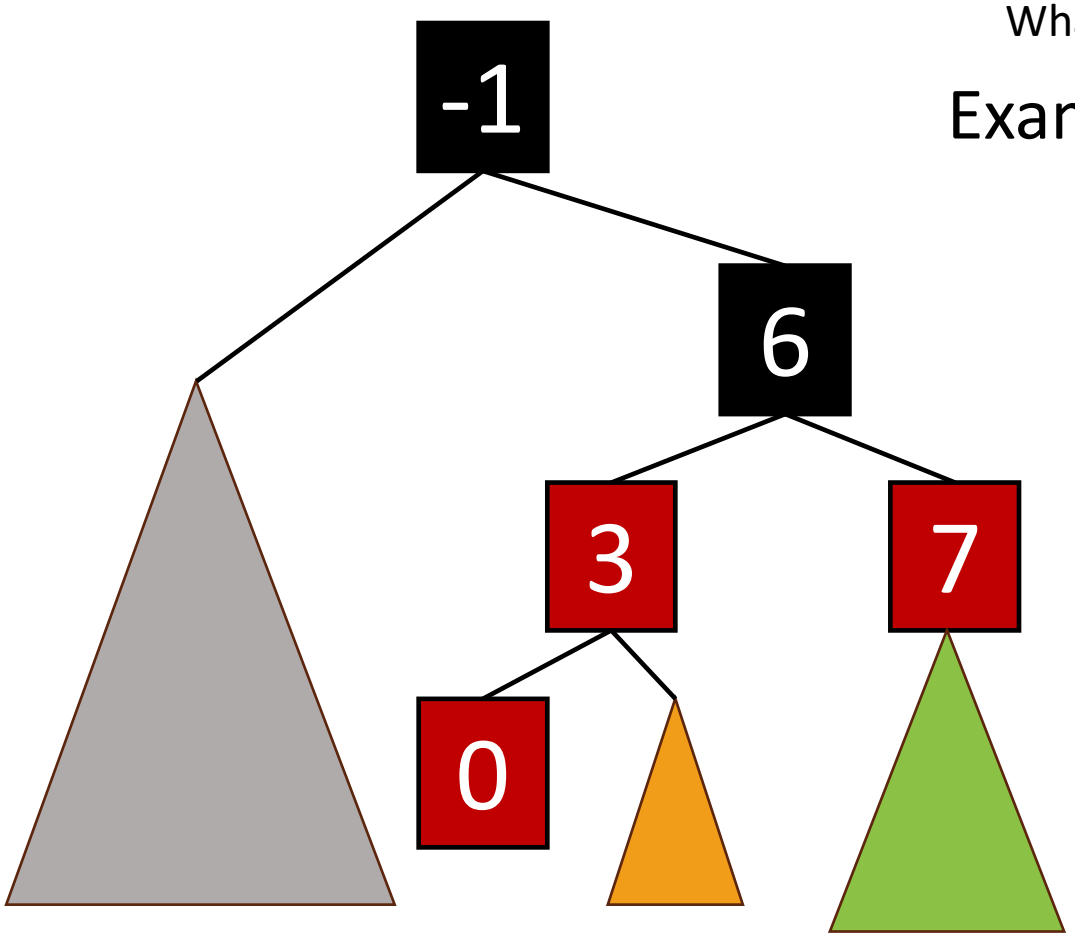

# We need a bit more context

- Add 0 as a red node.



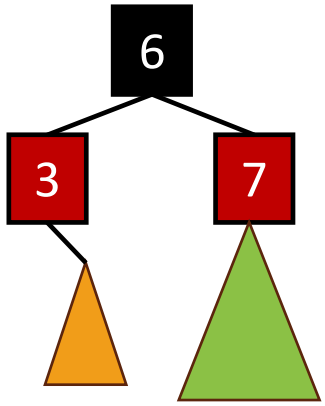

What if it looks like this?

Example: insert 0



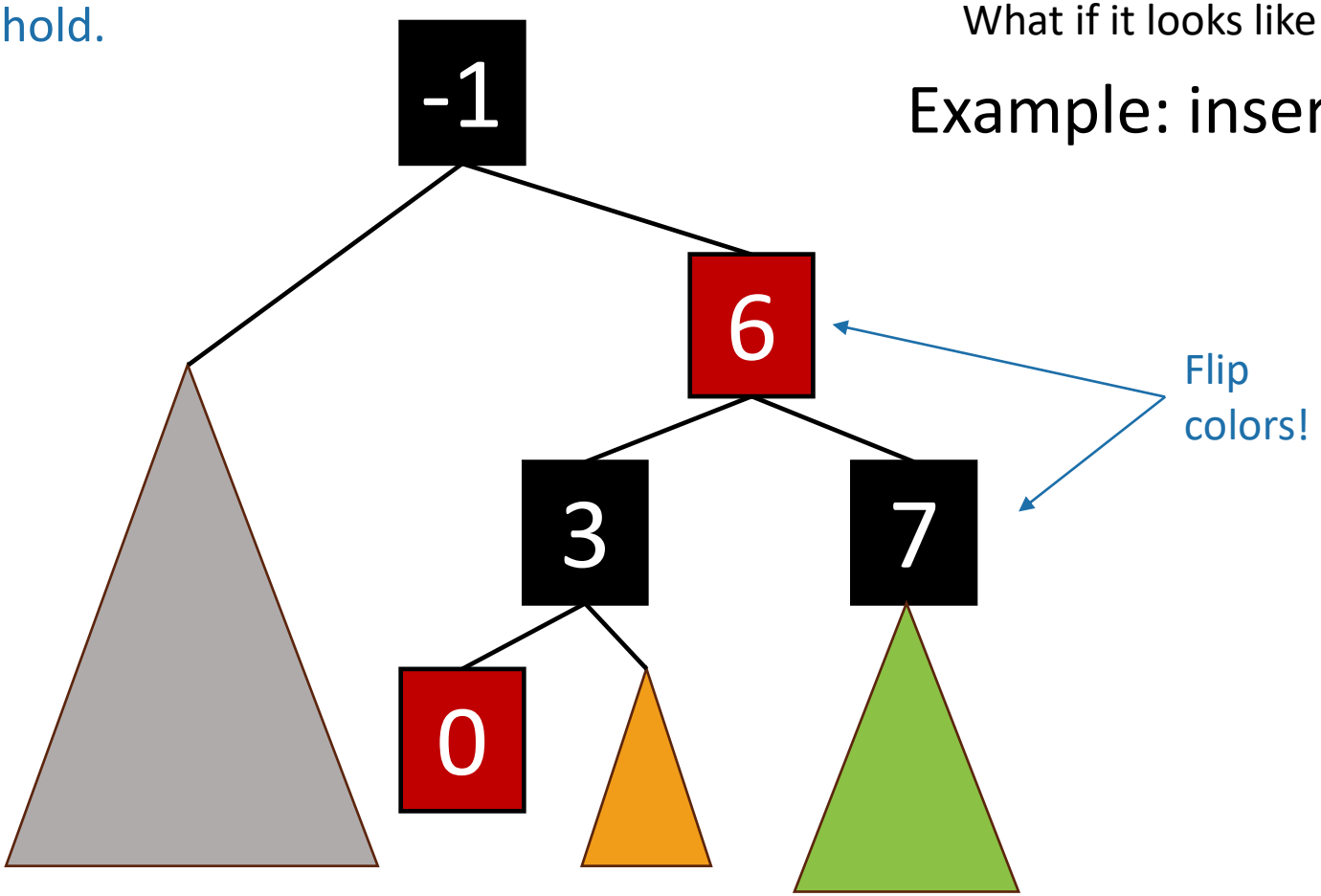

# We need a bit more context

- Add 0 as a red node.
- **Claim:** RB-Tree properties still hold.

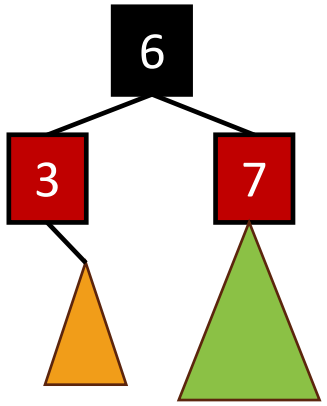



What if it looks like this?

Example: insert 0

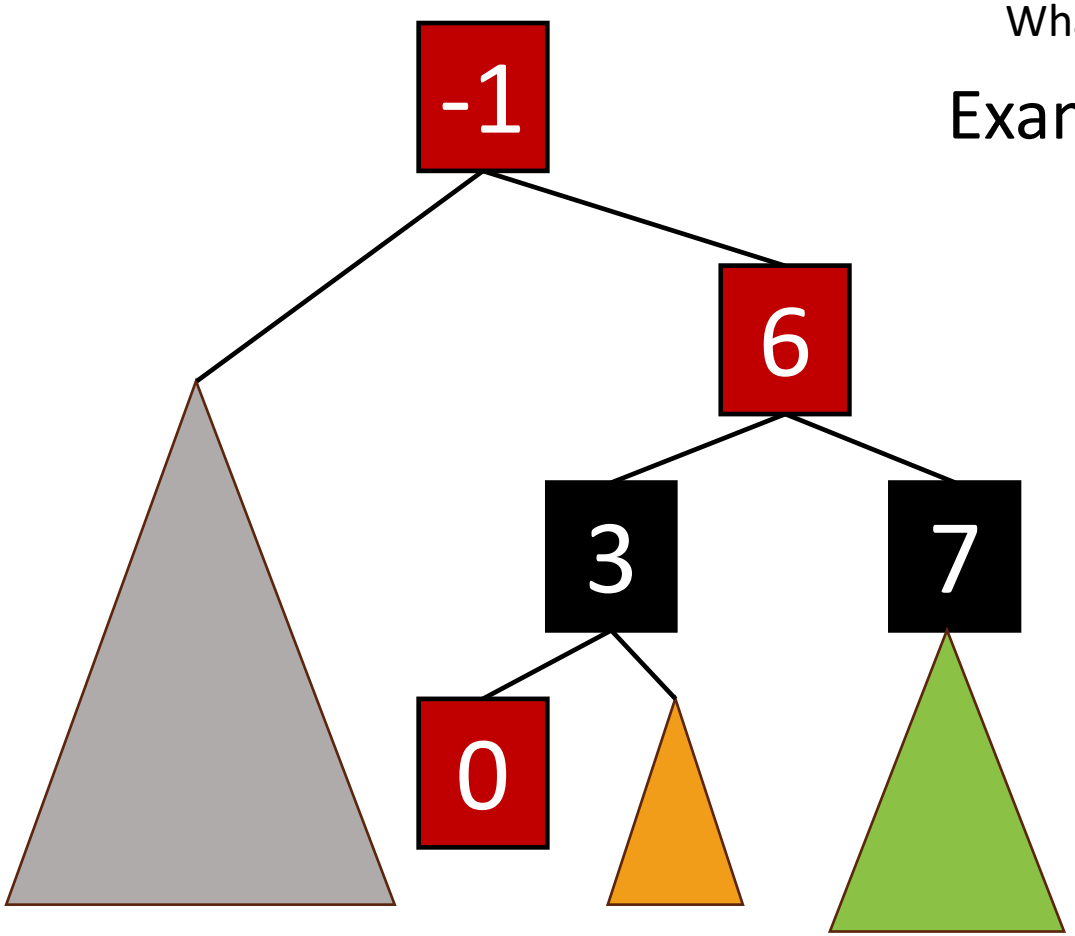



# But what if that was red?

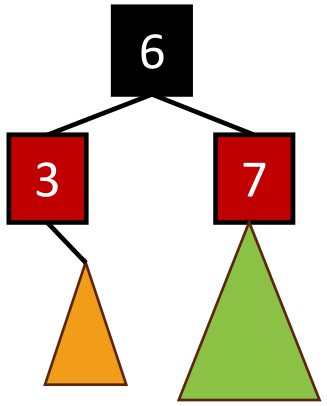



What if it looks like this?

Example: insert 0

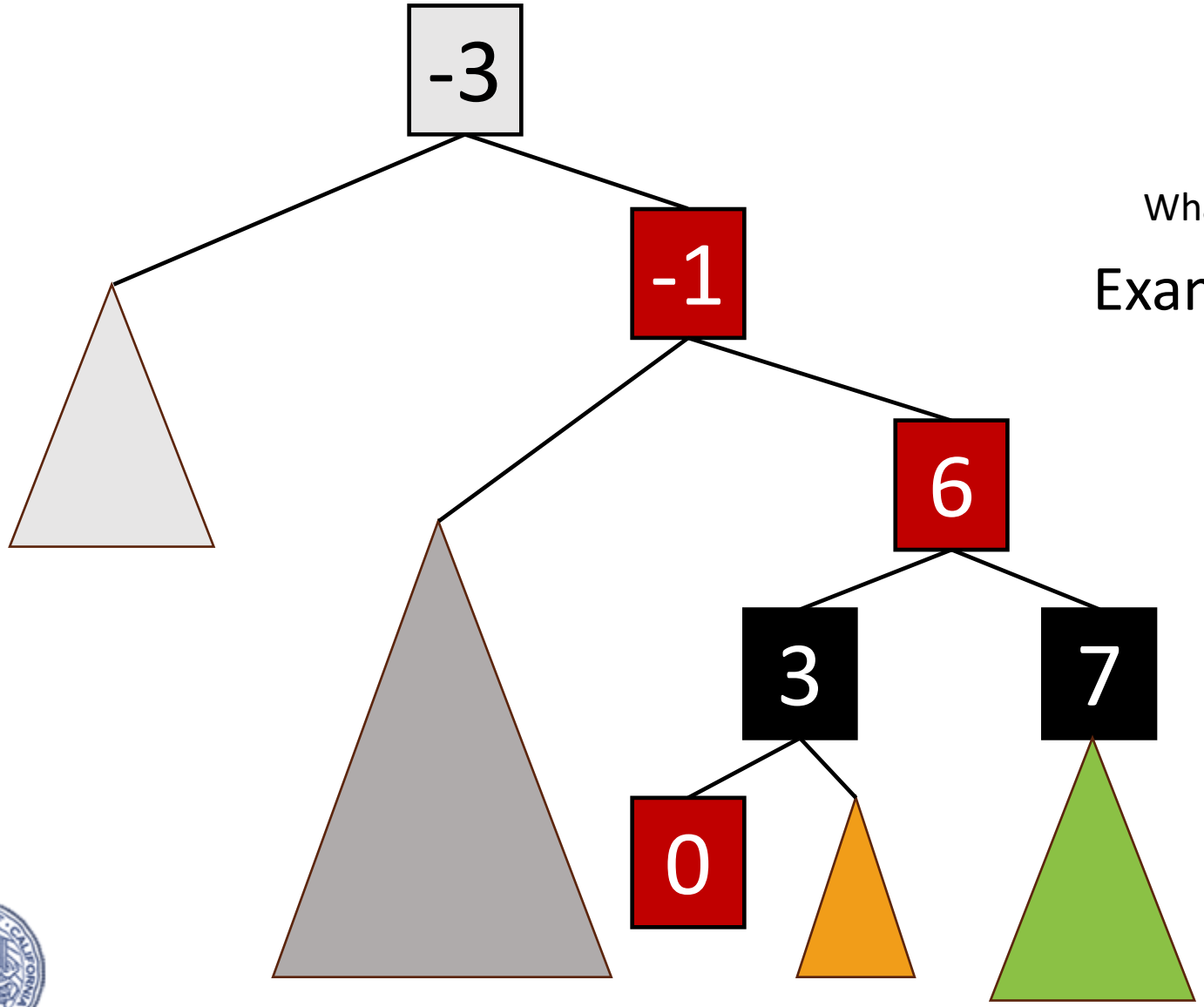



# More context…

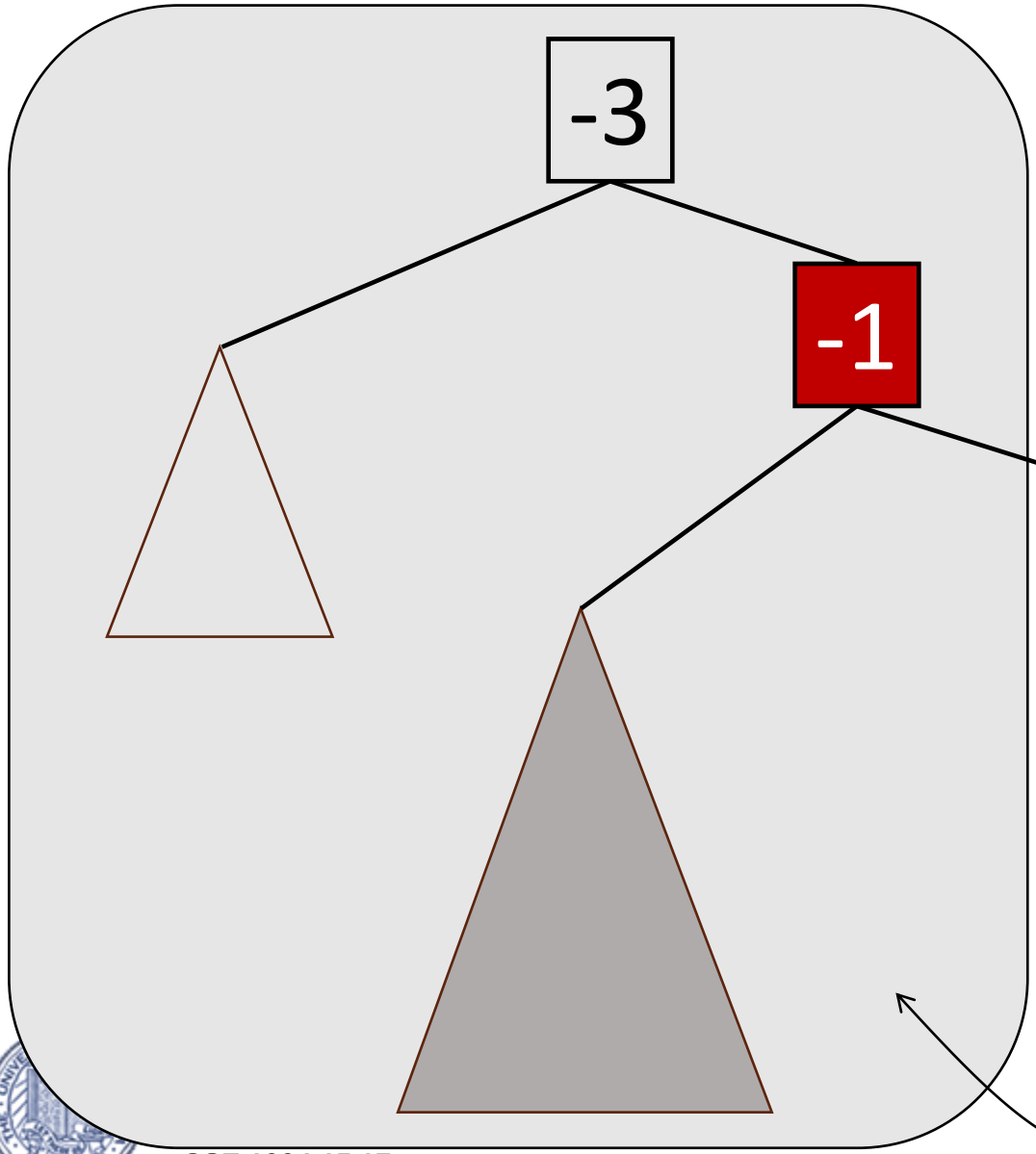


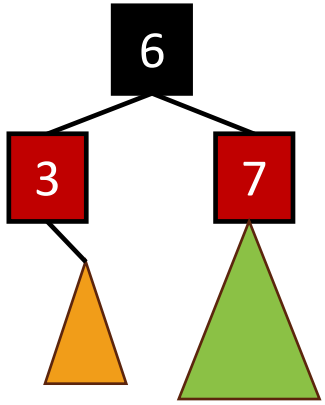

What if it looks like this?

Example: insert 0




# More context…



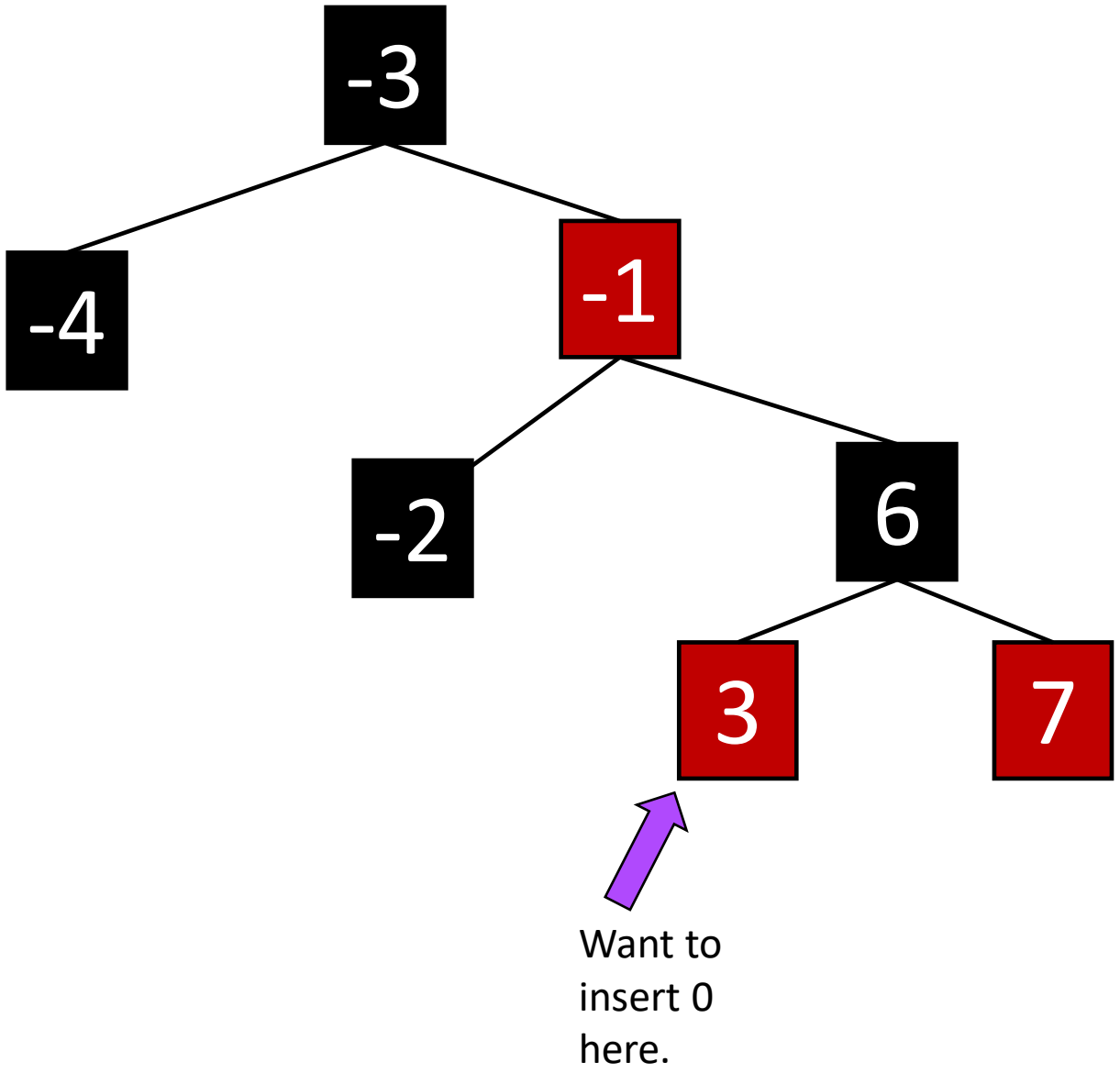

What if it looks like this?

Example: insert 0

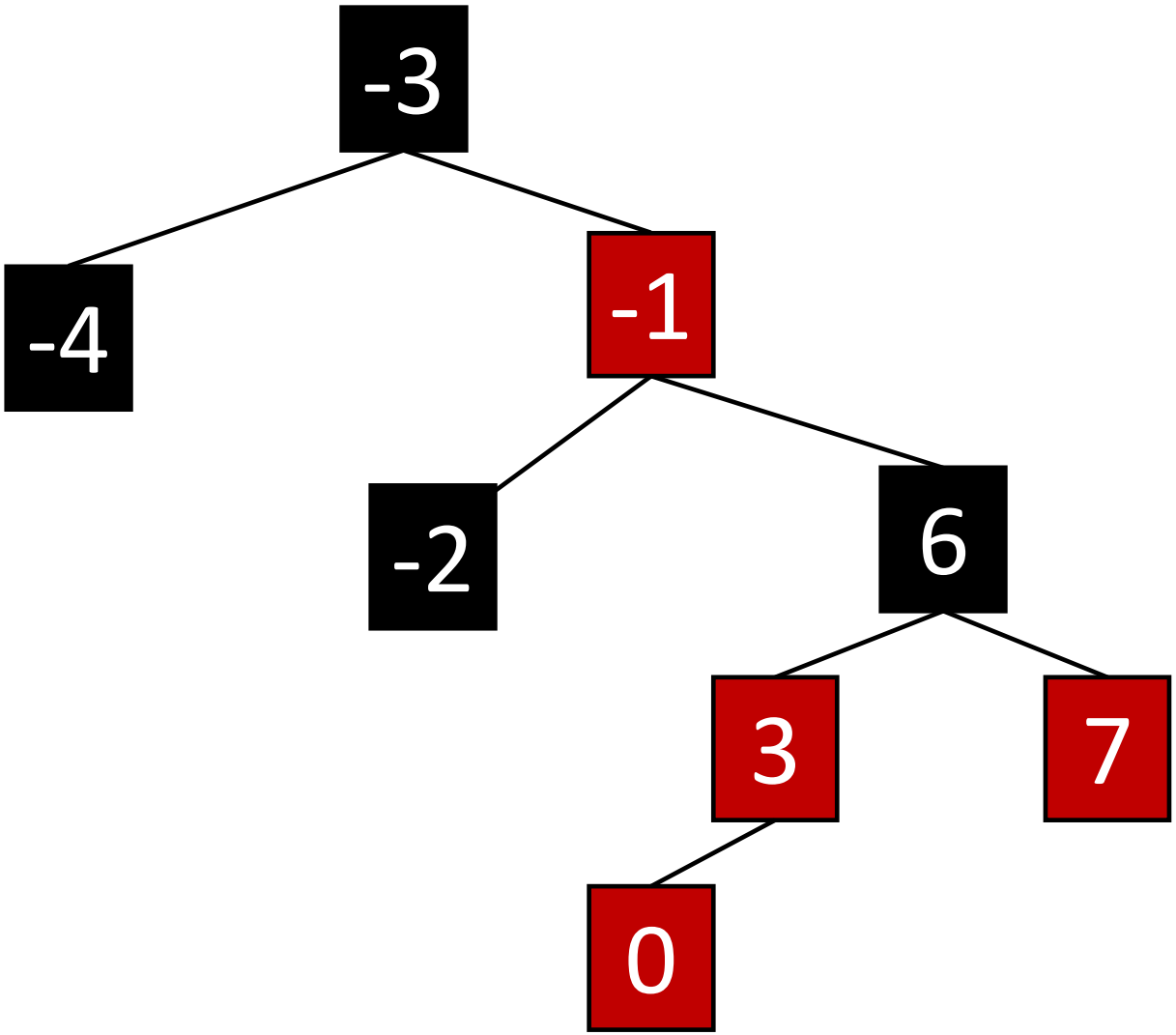
Now we're basically inserting 6 into some **smaller tree**. Recurse!

This one!

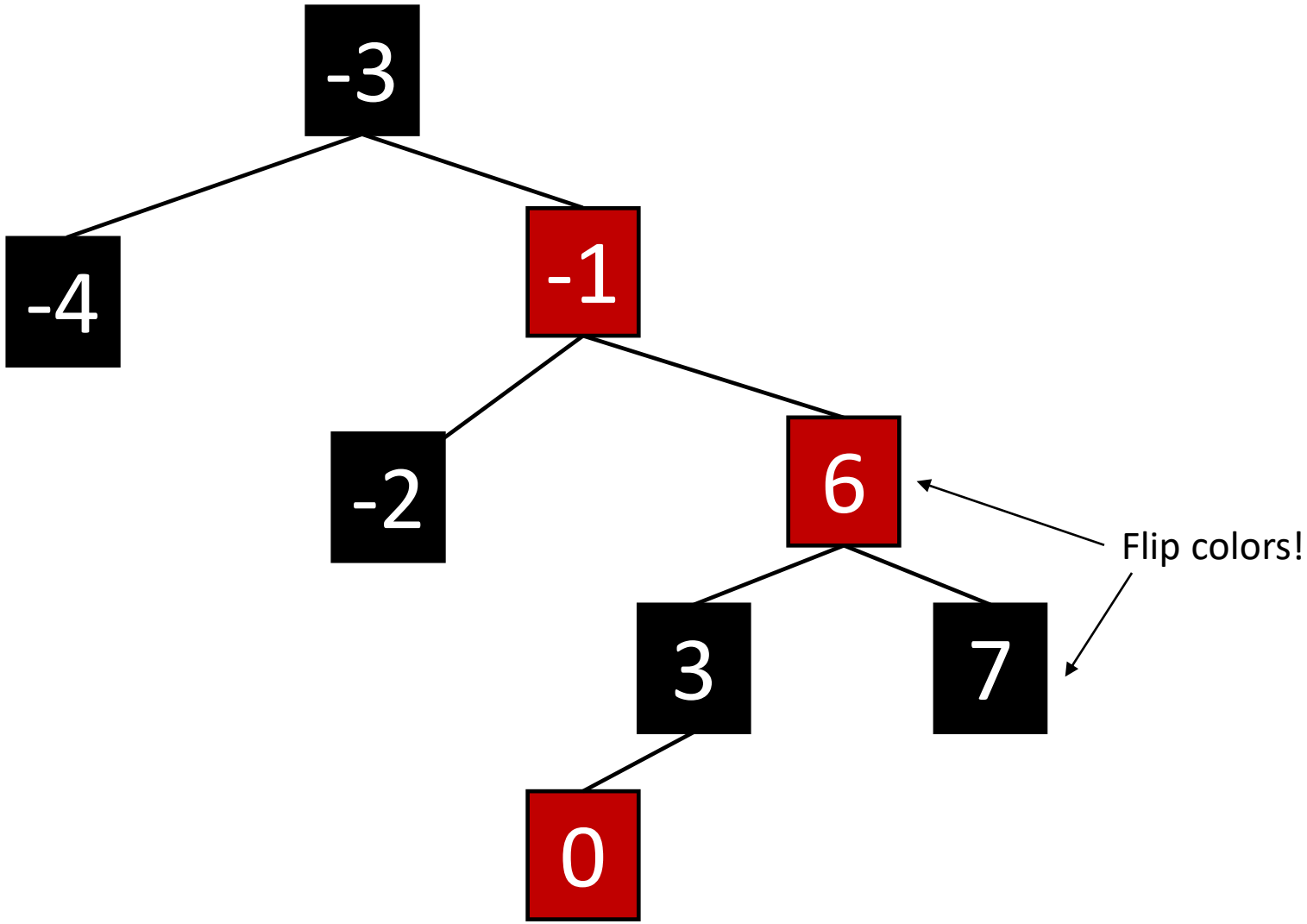
# Example, part I



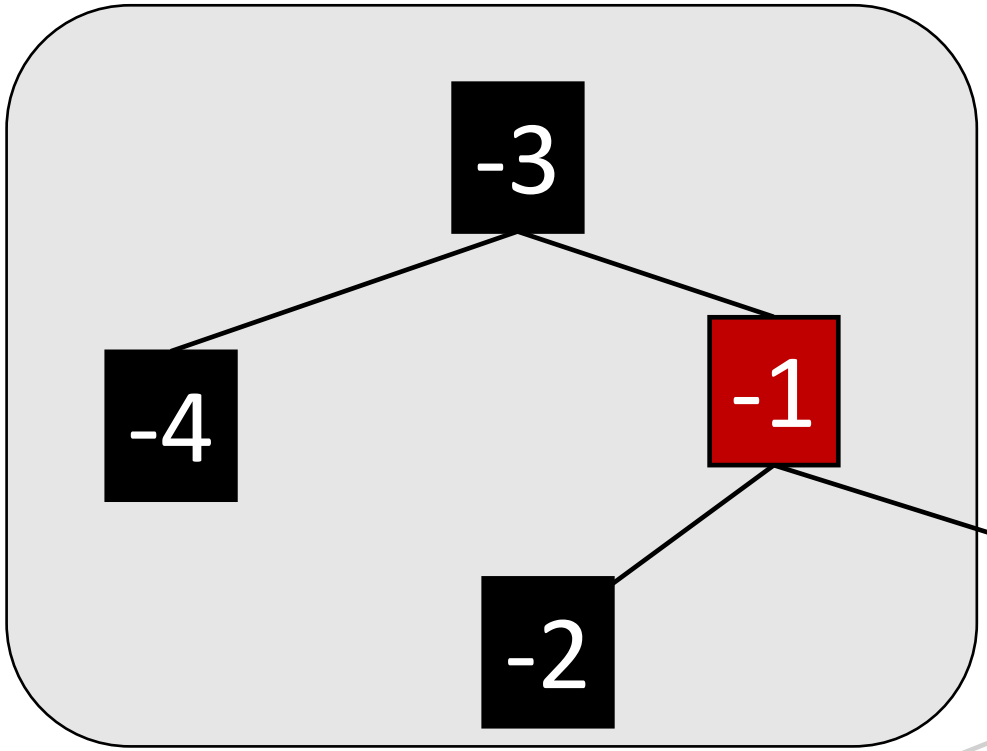

# Example, part I



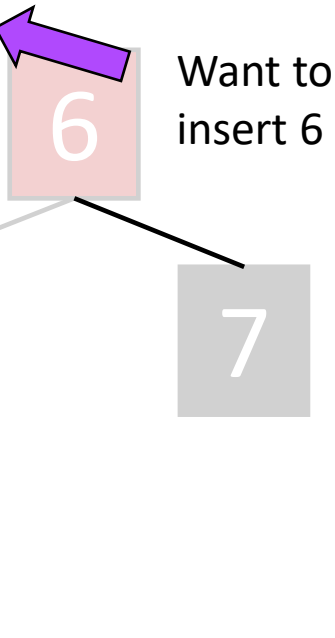

# Example, part I




# Example, part I



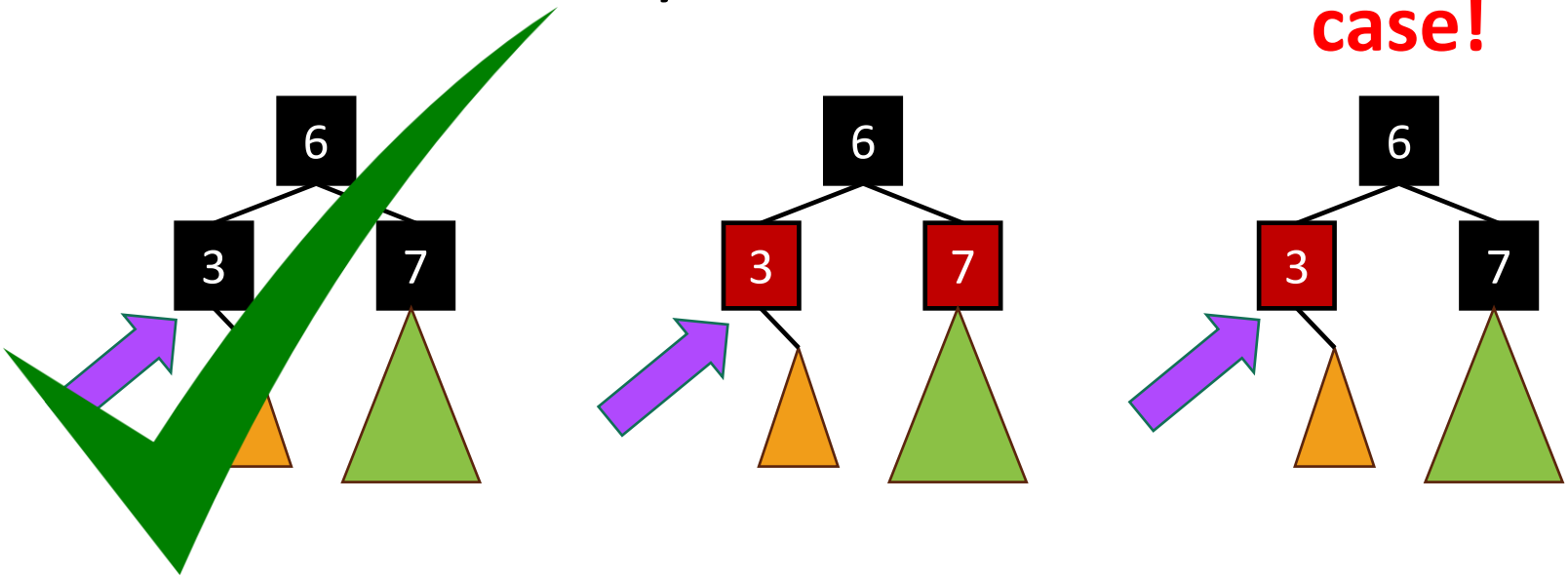

Need to know how to insert into trees that look like this…




Want to insert 6 here.

# INSERT: Many cases

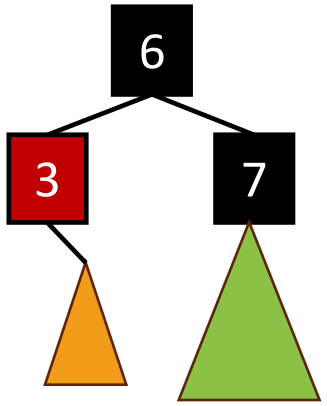
**That's this case!**



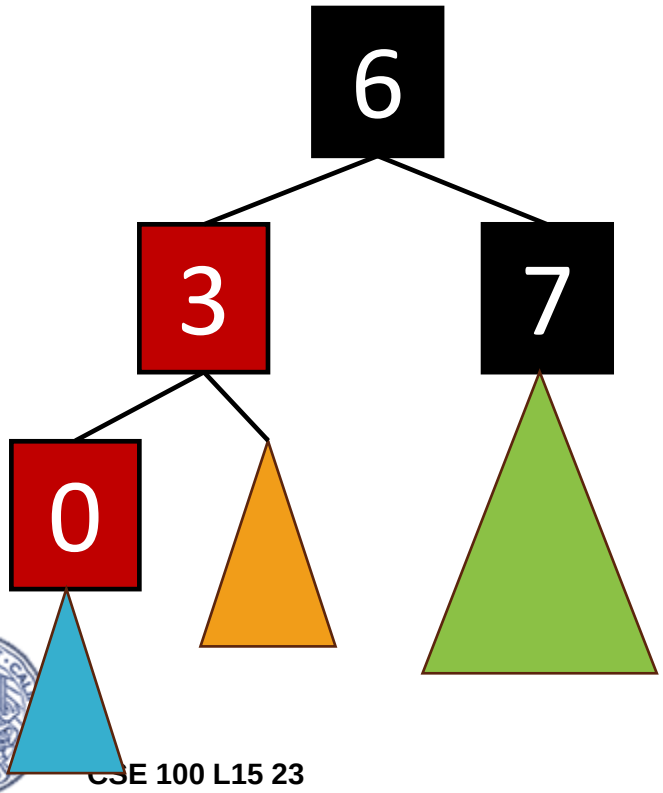

- Suppose we want to insert 0 **here**.
- There are 3 "important" cases for different colorings of the existing tree, and there are 9 more cases for all of the various symmetries of these 3 cases.

# INSERT: Case 3

- Make a new **red node**.
- Insert it as you would normally.
- Fix things up if needed.




What if it looks like this?

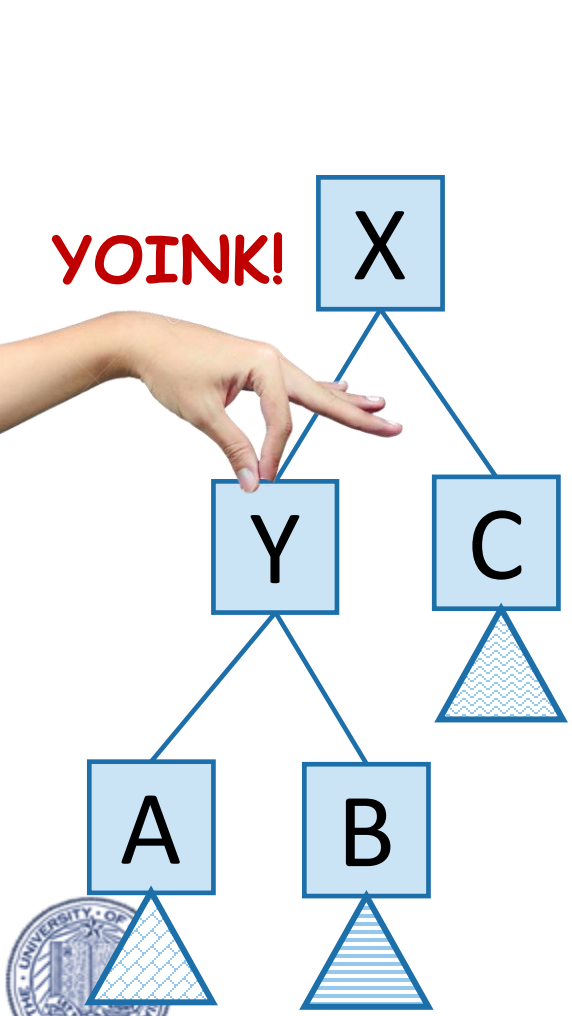


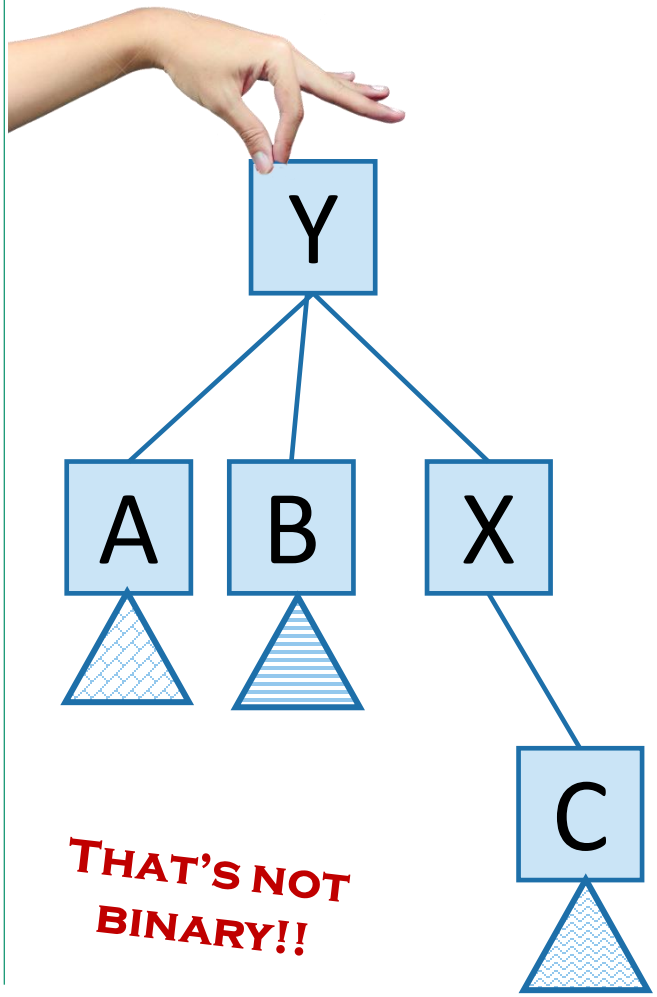

Example: Insert 0.

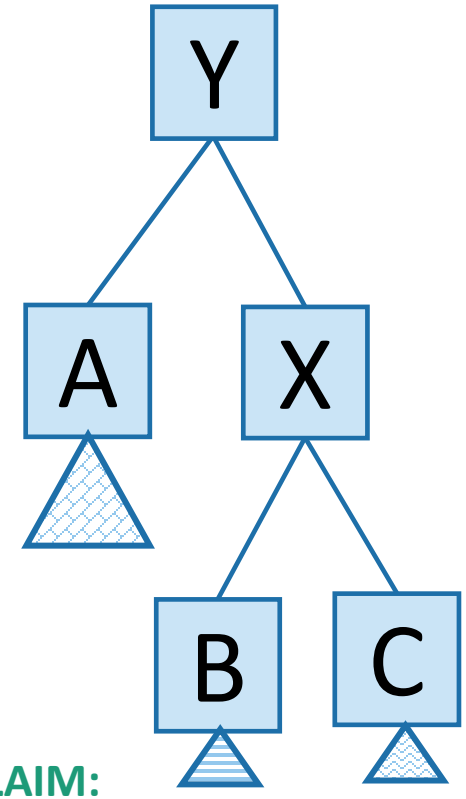
- Maybe with a subtree below it.

# Recall Rotations

- Maintain Binary Search Tree (BST) property, while moving stuff around.



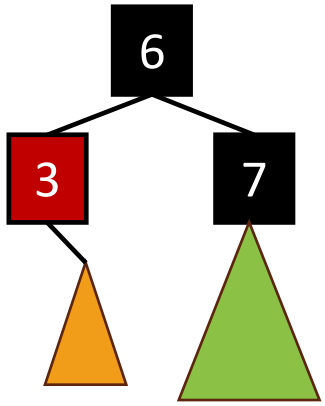




**CLAIM:**
this still has BST property.

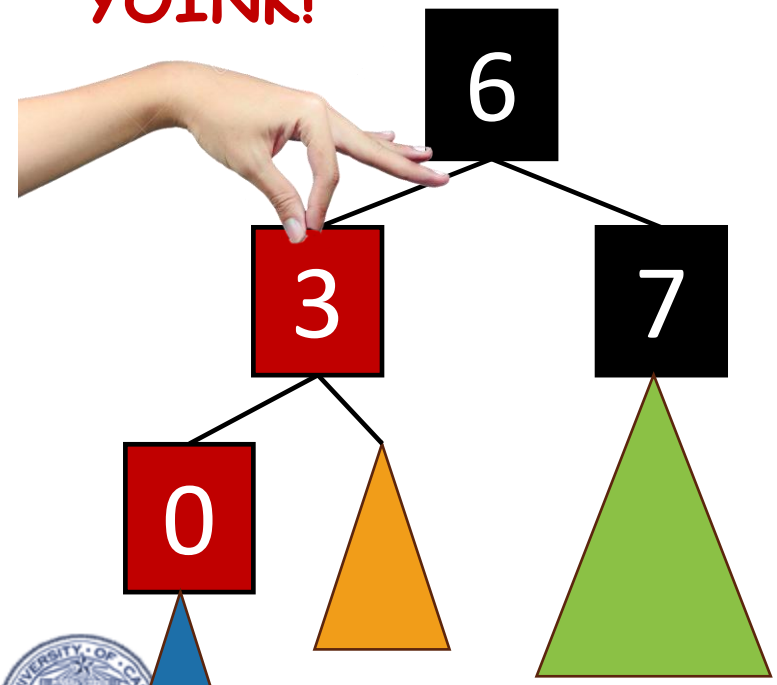
# Inserting into a Red-Black Tree

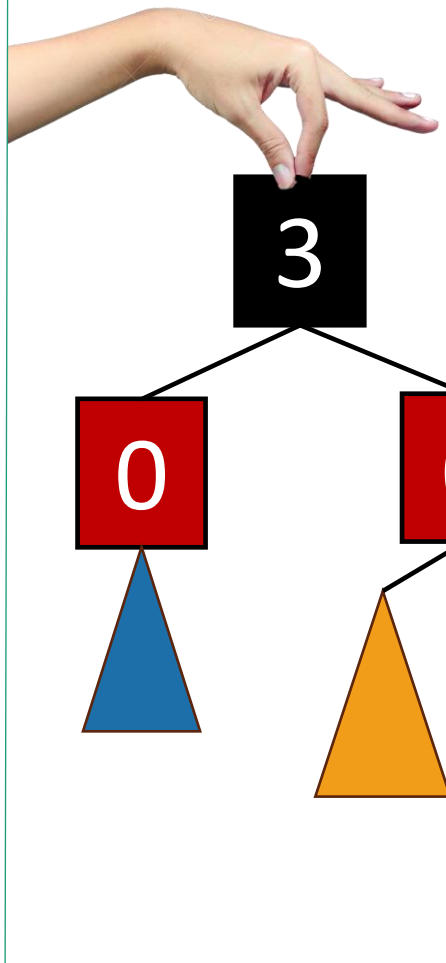
- Make a new **red node**.
- Insert it as you would normally.
- Fix things up if needed.




What if it looks like this?

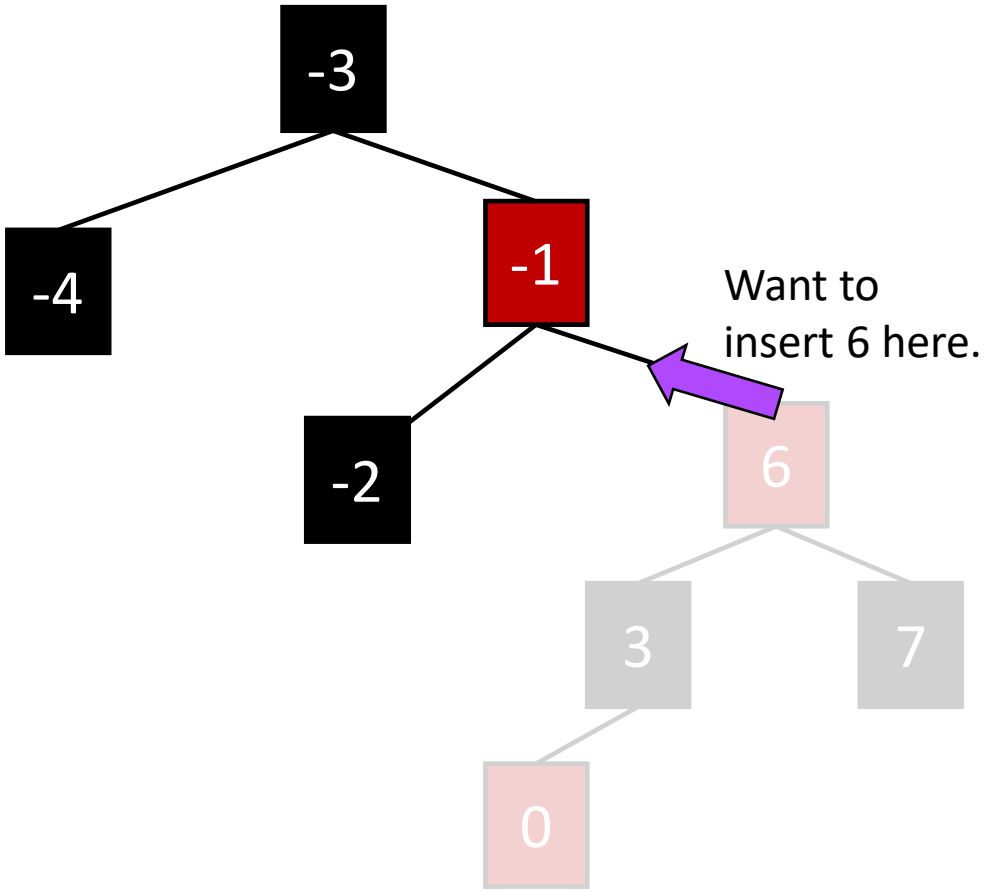
**YOINK!**



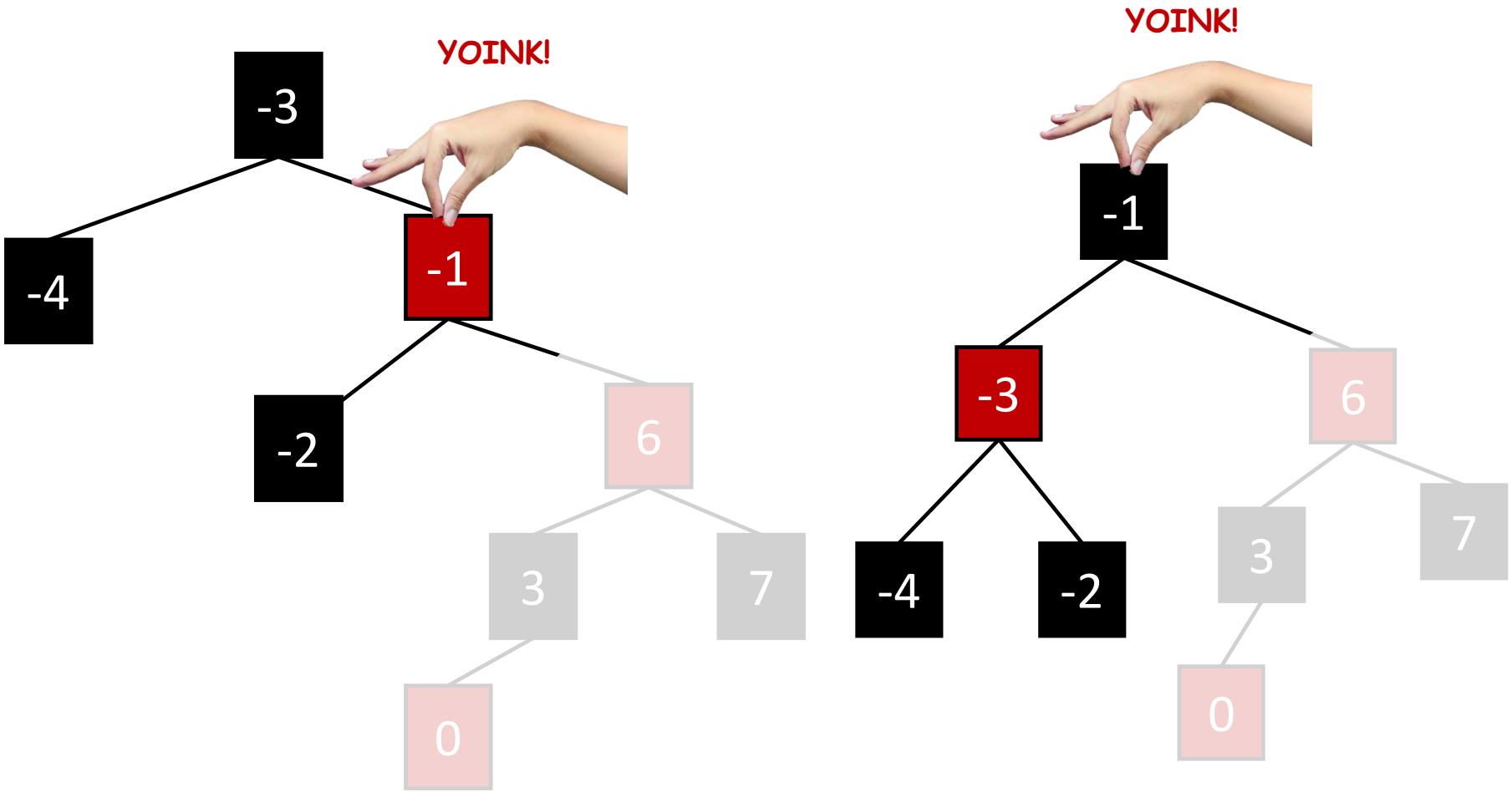




Argue that this is a good thing to do!

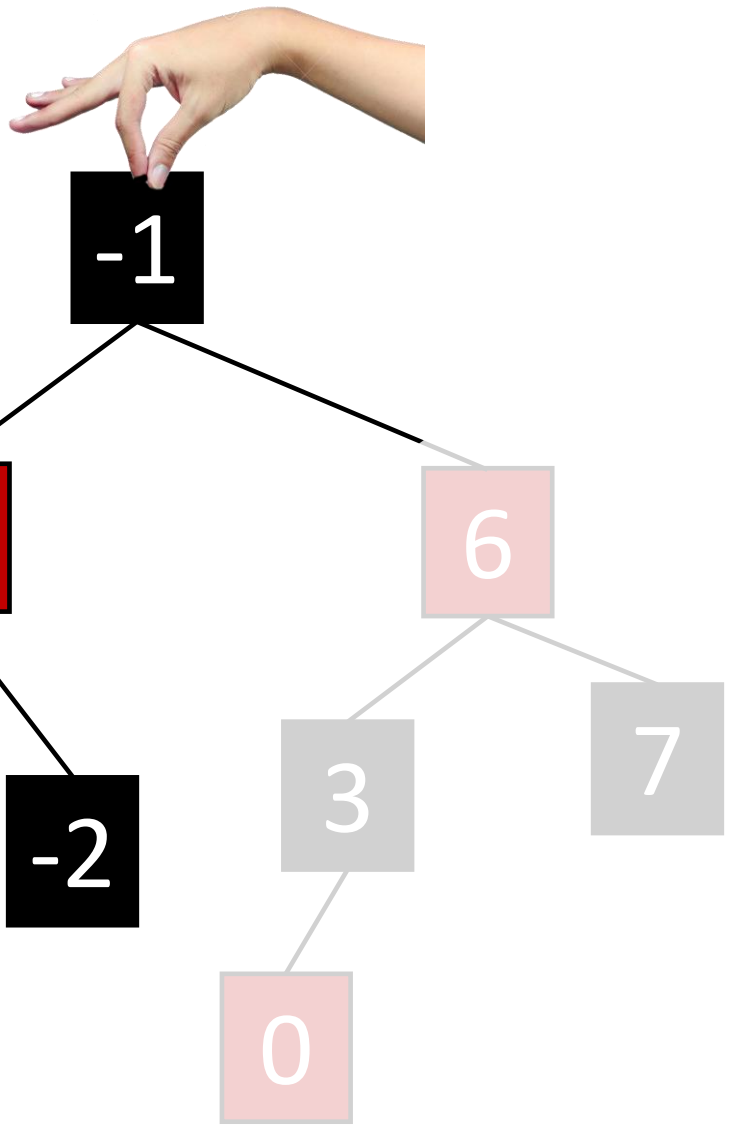
# Example, part 2



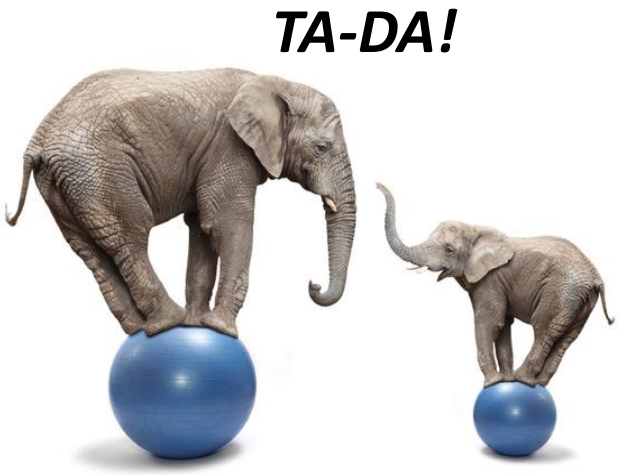

# Example, part 2

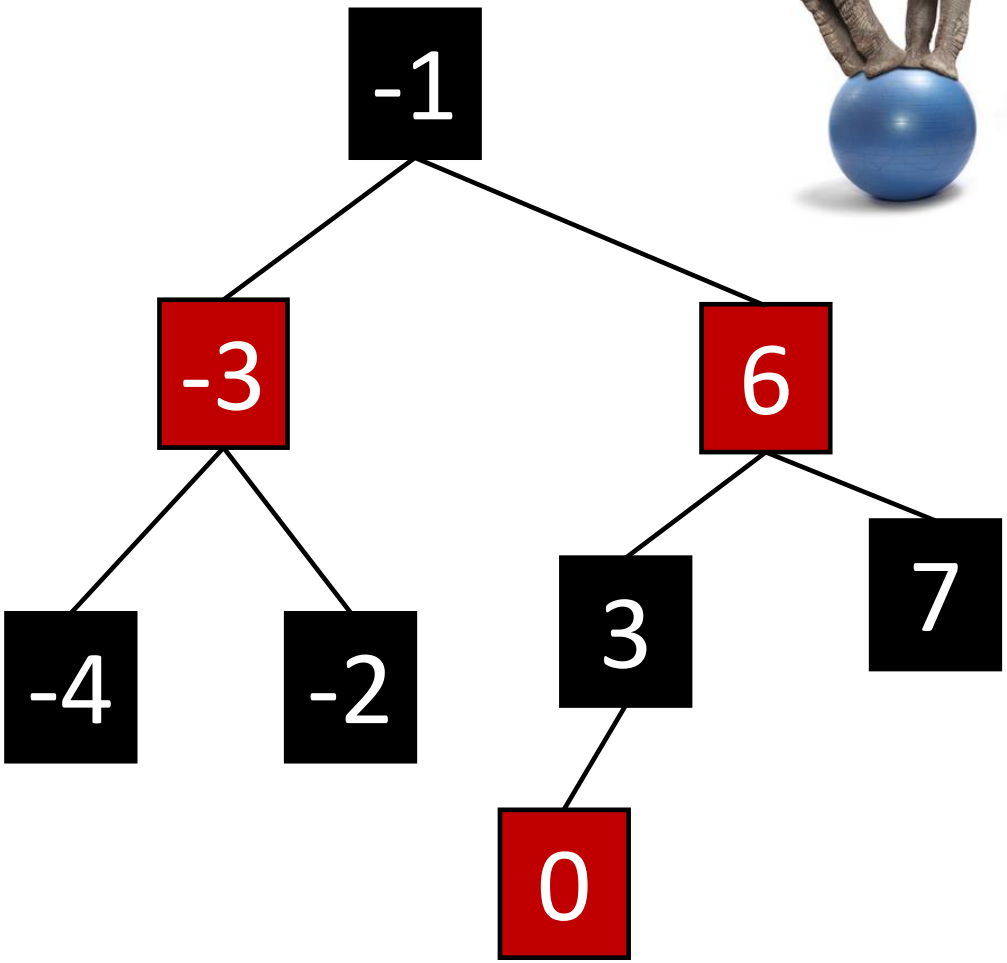



# Example, part 2

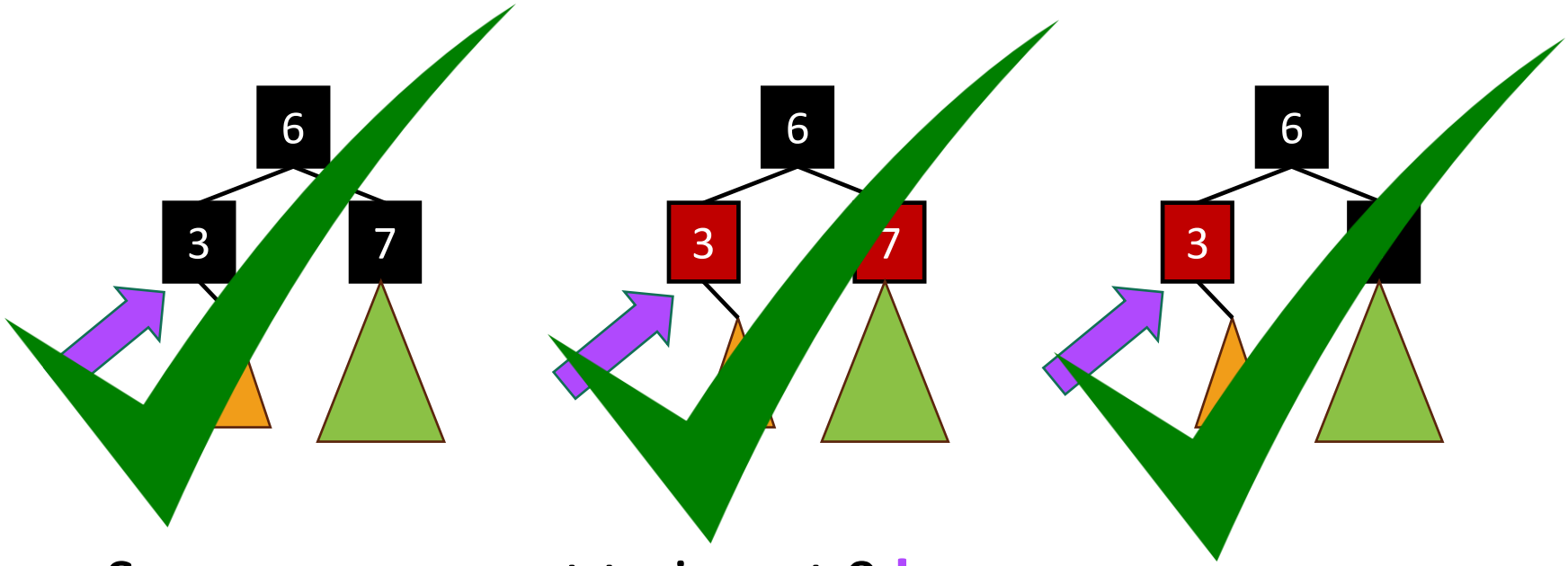



# Example, part 2



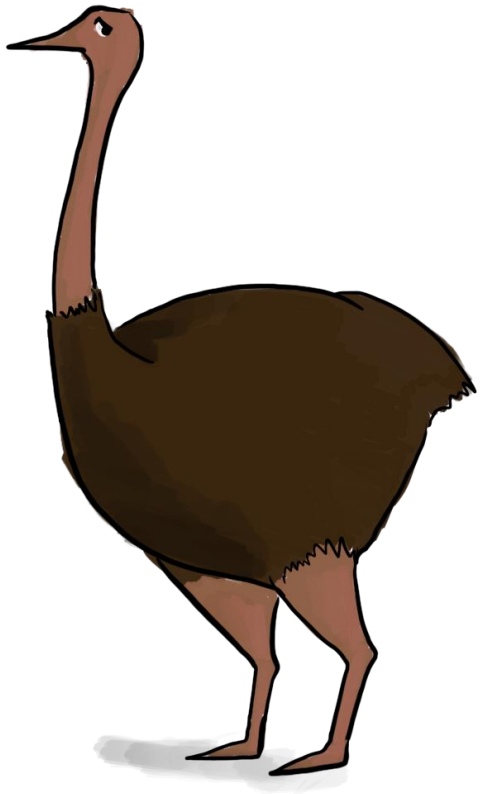

# Many cases




- Suppose we want to insert 0 **here**.
- There are 3 “important” cases for different colorings of the existing tree, and there are 9 more cases for all of the various symmetries of these 3 cases.

# Deleting from a Red-Black tree

**Fun exercise!**



Ollie the over-achieving ostrich

# That's a lot of cases!

- You are **not responsible** for the nitty-gritty details of Red-Black Trees. (For this class)
  - Though implementing them is a great exercise!
- You should know:
  - What are the properties of an RB tree?
  - And (more important) why does that guarantee that they are balanced?

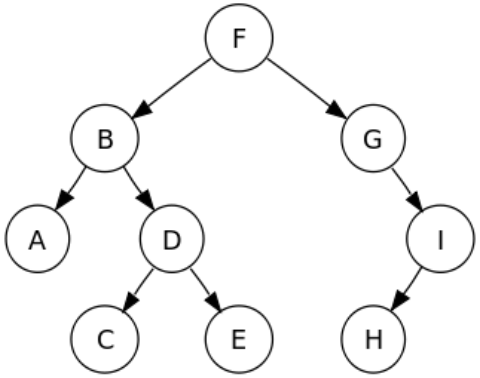
# What have we learned?

- Red-Black Trees always have height at most 2log(n+1).
- As with general Binary Search Trees, all operations are O(height)
- So all operations with RBTrees are O(log(n)).

# Conclusion: The best of both worlds

| | Sorted Arrays | Linked Lists | Binary Search Trees* |
|---|---|---|---|
| Search | O(log(n)) | O(n) | O(log(n)) |
| Delete | O(n) | O(n) | O(log(n)) |
| Insert | O(n) | O(1) | O(log(n)) |

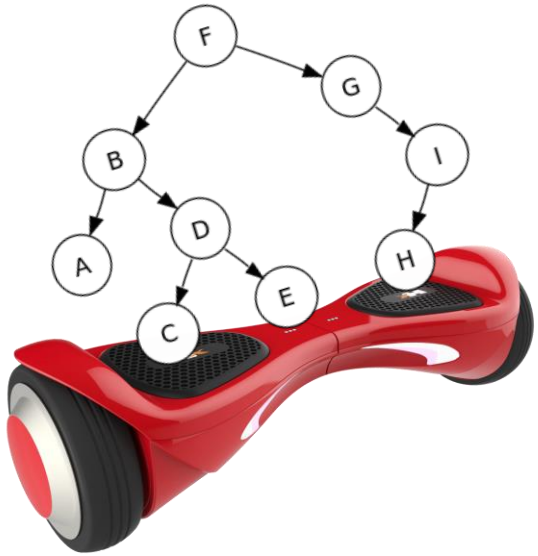
# Today (part 1)




- Begin a brief foray into data structures!
- Binary search trees
  - You may remember these from CSE 30
  - They are better when they're balanced.

this will lead us to…

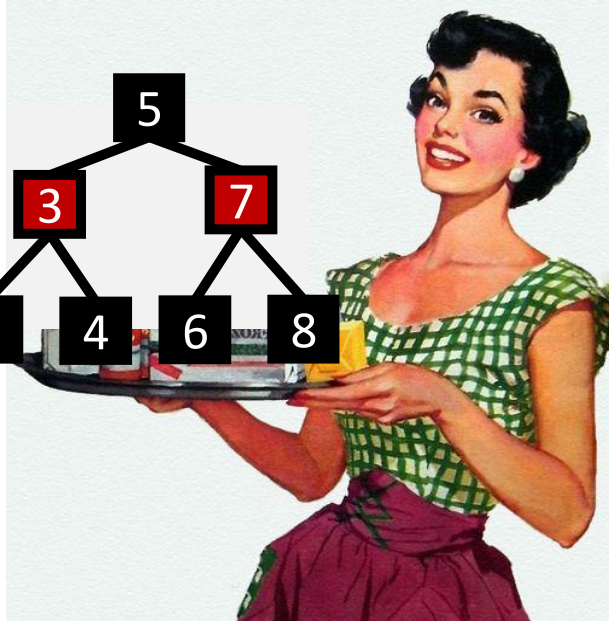
- Self-Balancing Binary Search Tre
  - **Red-Black** trees.




Recap

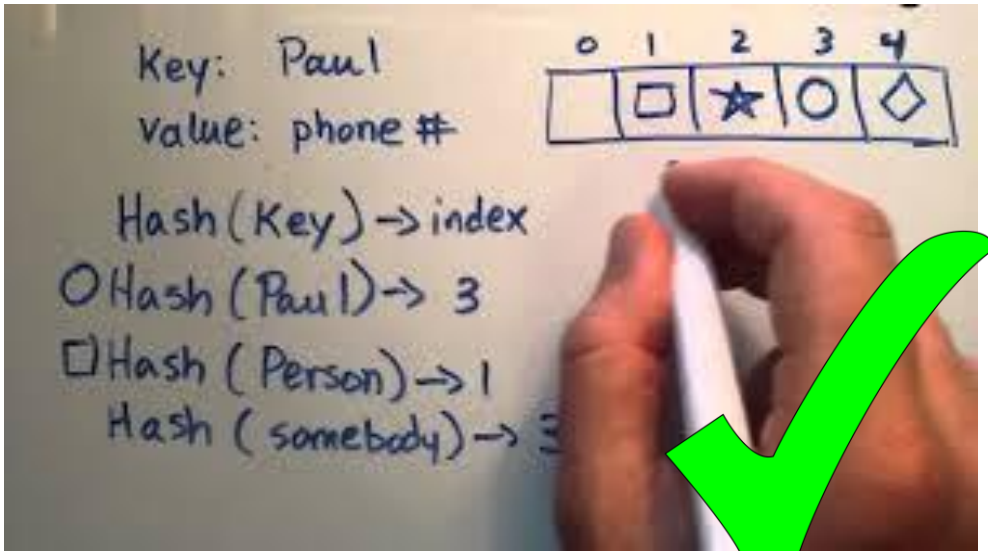
# Recap

- Balanced binary trees are the best of both worlds!
- But we need to keep them balanced.
- **Red-Black Trees** do that for us.
  - We get O(log(n))-time INSERT/DELETE/SEARCH
  - Clever idea: have a proxy for balance

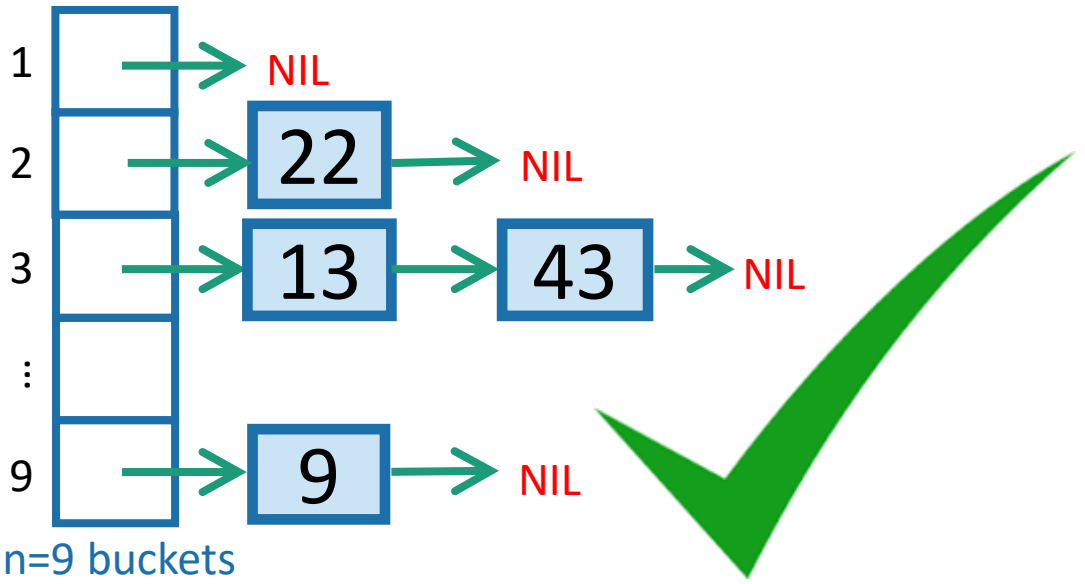


# Next Part

- **Hashing!**




# Today (part 2): hashing

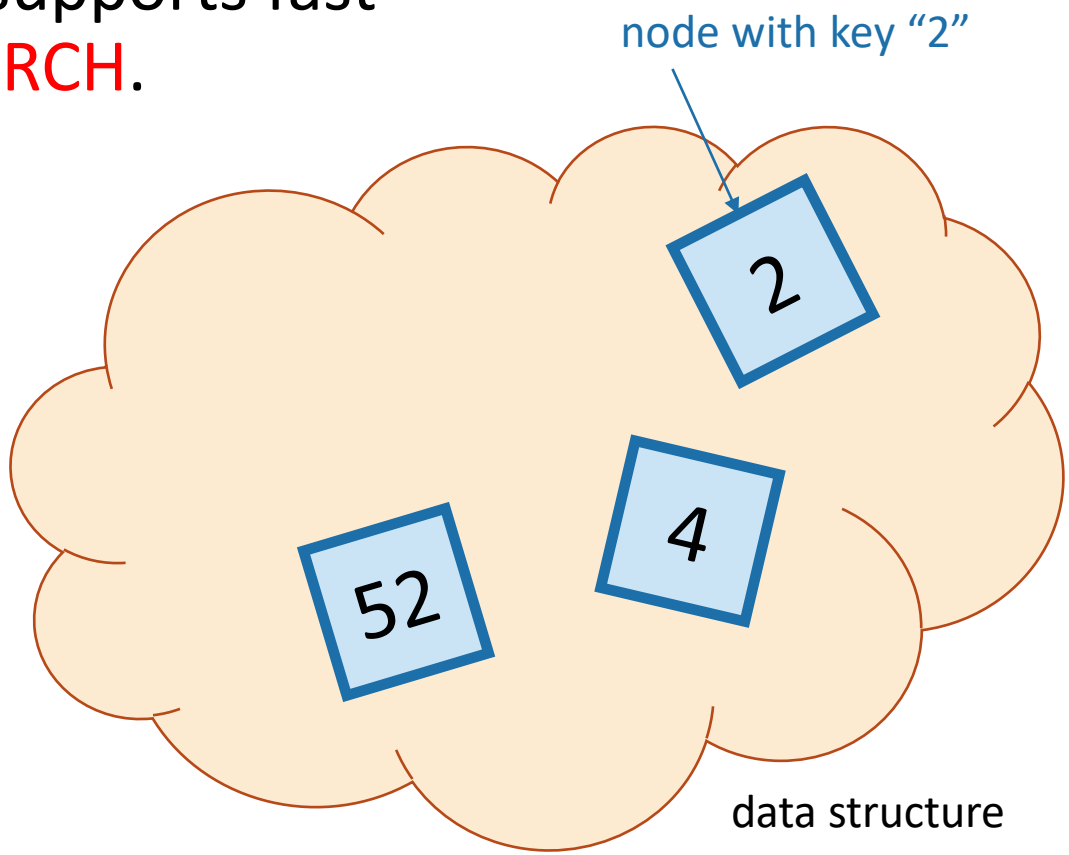


# Outline

- Hash tables are another sort of data structure that allows fast INSERT/DELETE/SEARCH.
  - like self-balancing binary trees
  - The difference is we can get better performance in expectation by using randomness.
- **Hash families** are the magic behind hash tables.
- **Universal hash families** are even more magical.

# Goal:
## Just like last time

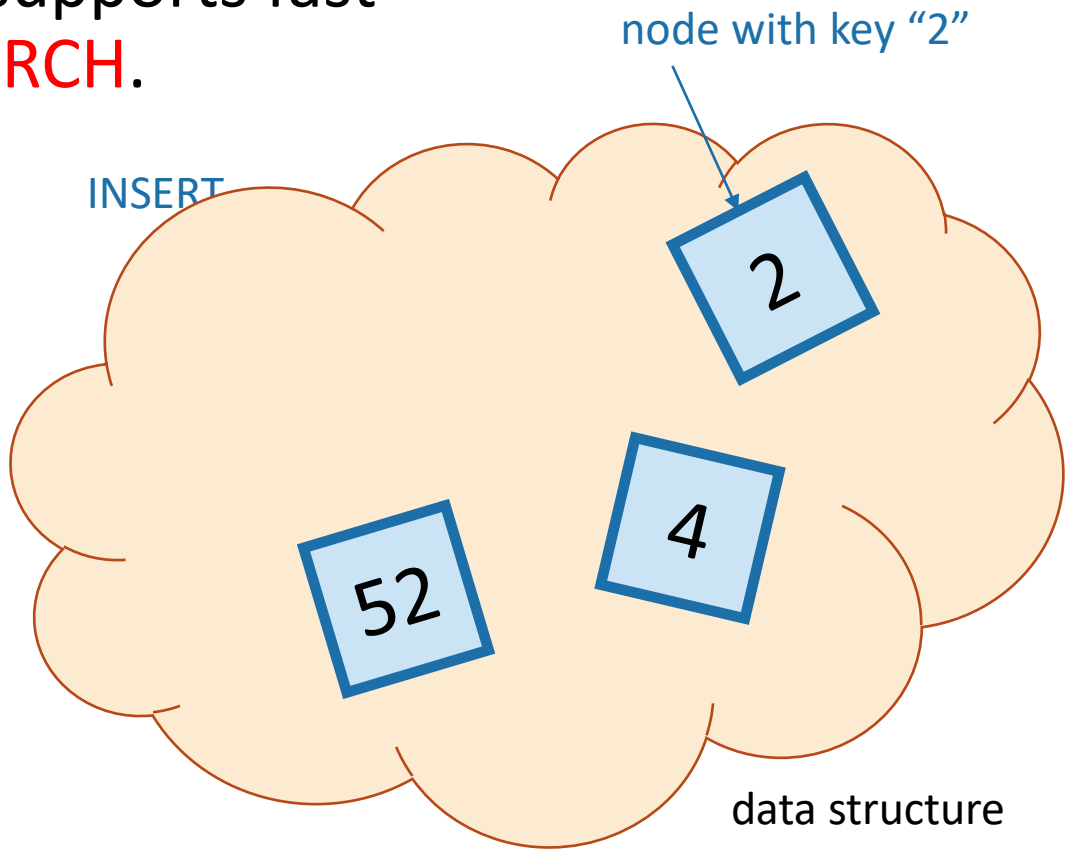
- We are interesting in putting nodes with keys into a data structure that supports fast INSERT/DELETE/SEARCH.




# Goal:
## Just like last time

- We are interesting in putting nodes with keys into a data structure that supports fast INSERT/DELETE/SEARCH.

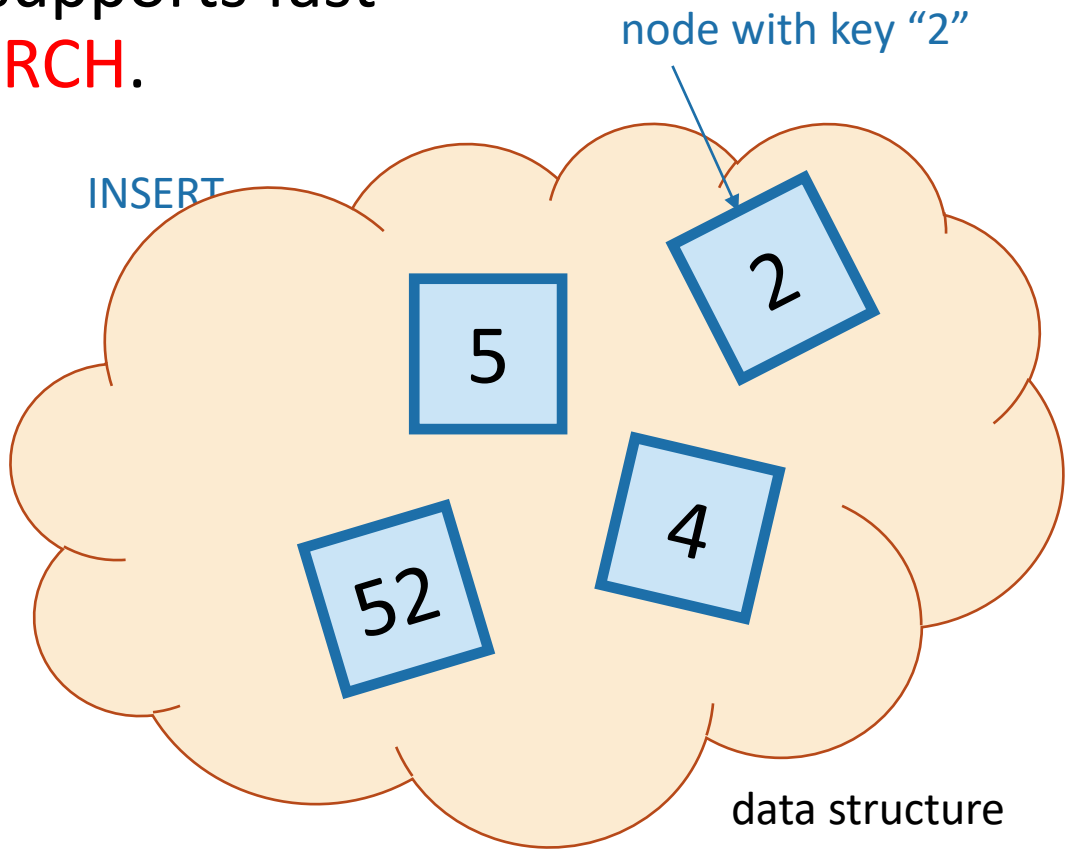
- INSERT 5




# Goal:
## Just like last time

- We are interesting in putting nodes with keys into a data structure that supports fast INSERT/DELETE/SEARCH.

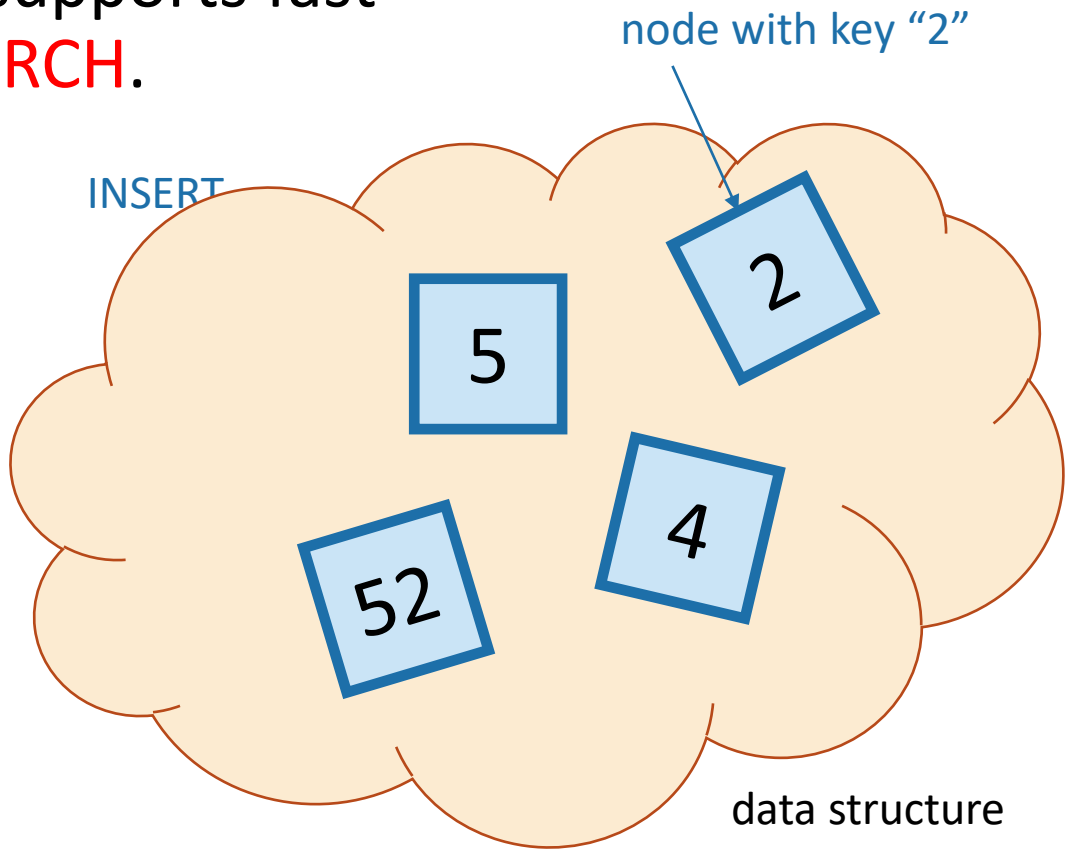
- INSERT 5




# Goal:
## Just like last time

- We are interesting in putting nodes with keys into a data structure that supports fast INSERT/DELETE/SEARCH.

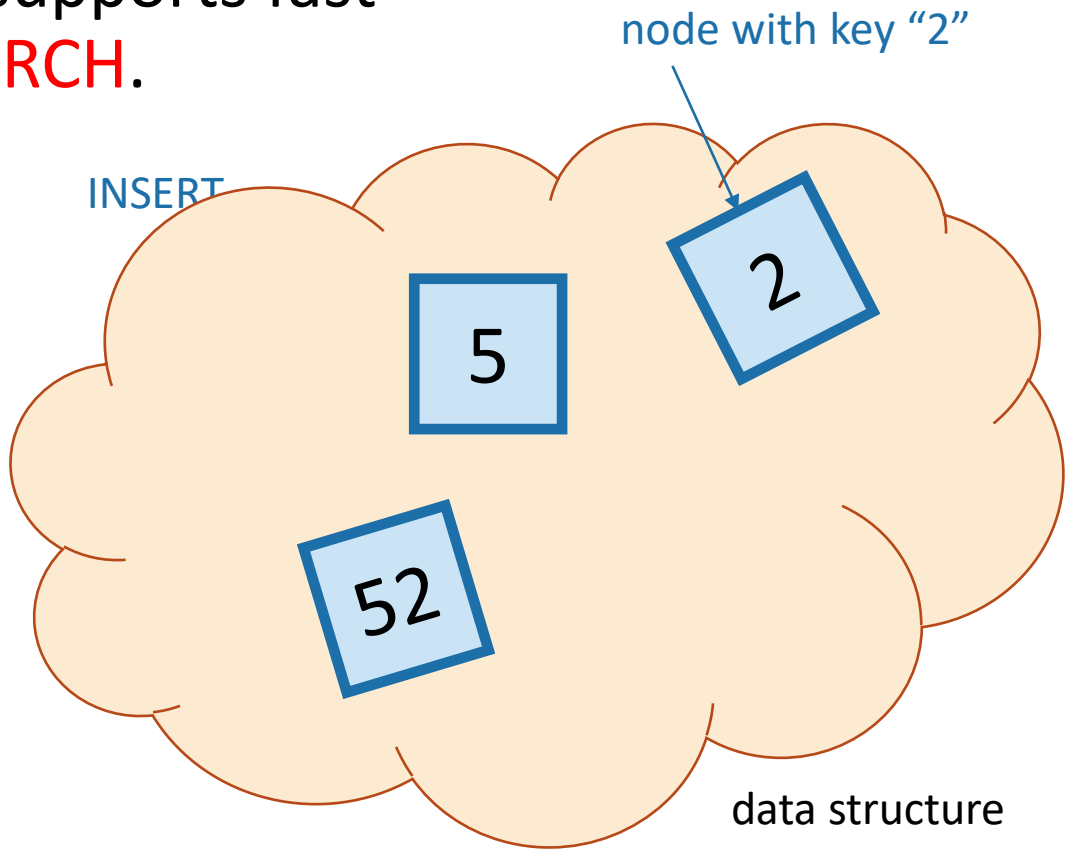
- INSERT 5
- DELETE 4




# Goal:
## Just like last time

- We are interesting in putting nodes with keys into a data structure that supports fast INSERT/DELETE/SEARCH.

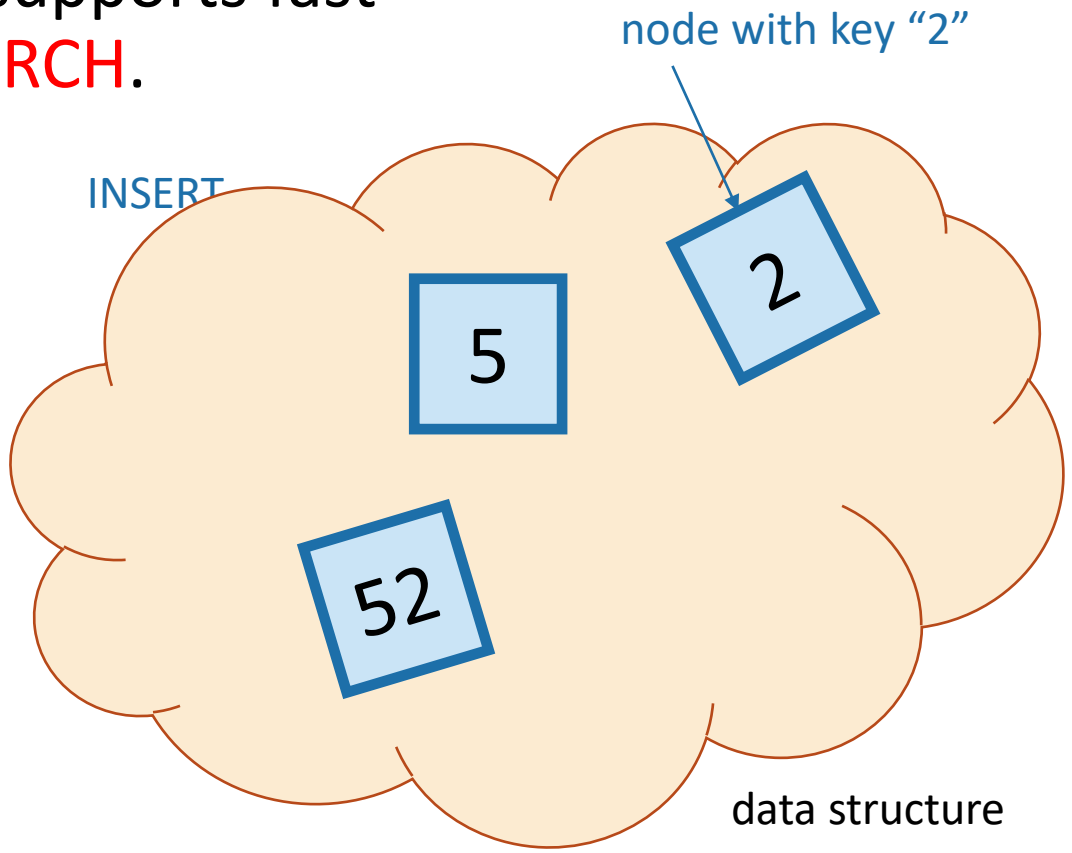
- INSERT 5
- DELETE 4




# Goal:
## Just like last time

- We are interesting in putting nodes with keys into a data structure that supports fast INSERT/DELETE/SEARCH.

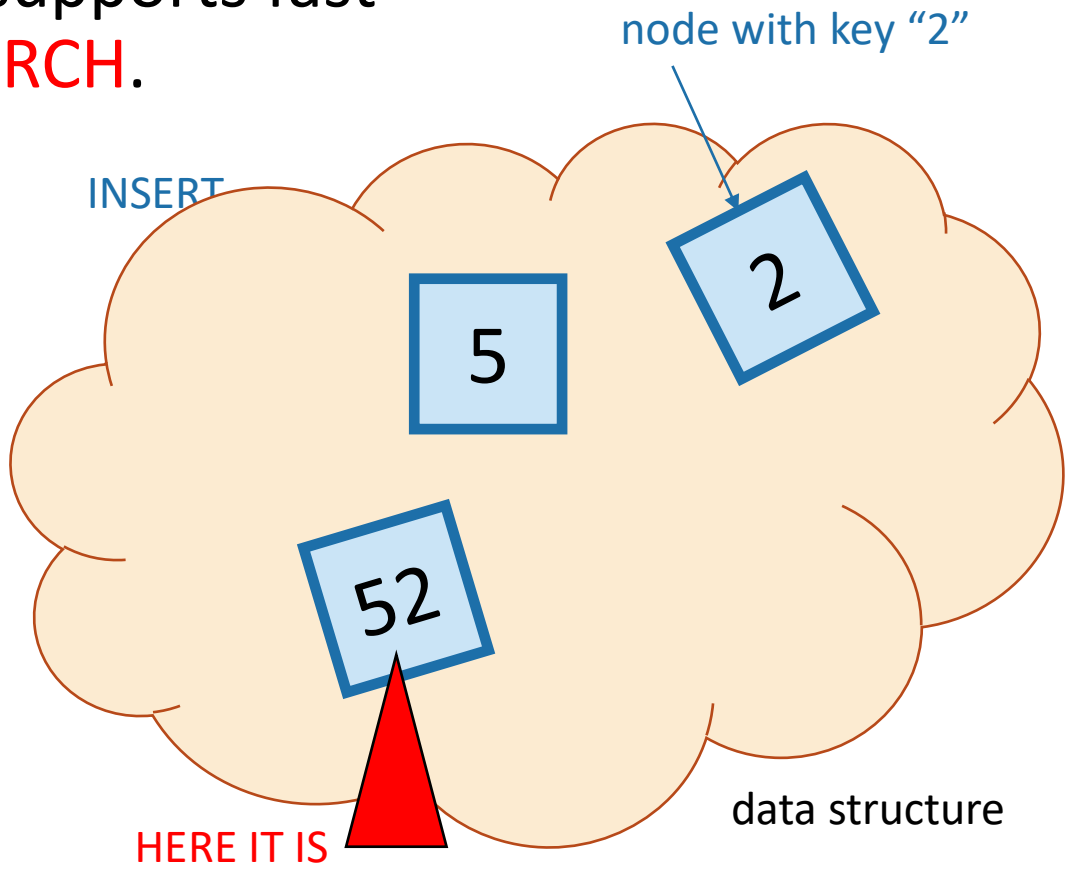
- INSERT 5
- DELETE 4
- SEARCH 52




# Goal:
## Just like last time

- We are interesting in putting nodes with keys into a data structure that supports fast INSERT/DELETE/SEARCH.

- INSERT 5
- DELETE 4
- SEARCH 52




# Last time

- Self balancing trees:
  - O(log(n)) deterministic INSERT/DELETE/SEARCH

#prettysweet

# Today:

- Hash tables:
  - O(1) expected time INSERT/DELETE/SEARCH
- Worse worst-case performance, but often great in practice.

#evensweeterinpractice

eg, Python's `dict`, Java's `HashSet/HashMap`, C++'s `unordered_map`

Hash tables are used for databases, caching, object representation, …

# One way to get O(1) time

This is called
"direct addressing"

# One way to get O(1) time

This is called "direct addressing"

- Say all keys are in the set {1,2,3,4,5,6,7,8,9}.

# One way to get O(1) time

This is called "direct addressing"

- Say all keys are in the set {1,2,3,4,5,6,7,8,9}.

# One way to get O(1) time

This is called "direct addressing"

- Say all keys are in the set {1,2,3,4,5,6,7,8,9}.


# One way to get O(1) time

This is called "direct addressing"

- Say all keys are in the set {1,2,3,4,5,6,7,8,9}.
- INSERT:


# One way to get O(1) time

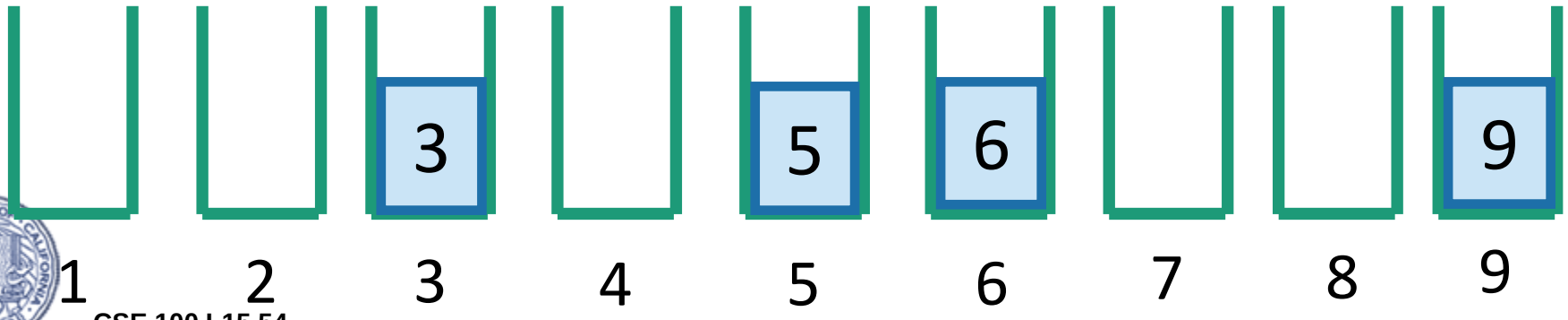
This is called "direct addressing"

- Say all keys are in the set {1,2,3,4,5,6,7,8,9}.
- INSERT: 9 6 3 5


# One way to get O(1) time

This is called "direct addressing"

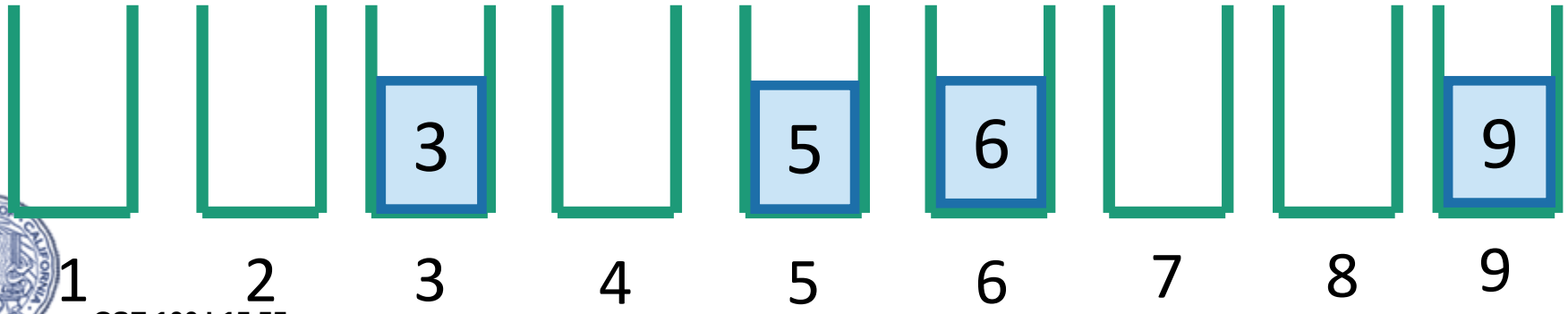
- Say all keys are in the set {1,2,3,4,5,6,7,8,9}.
- INSERT: 9 6 3 5



# One way to get O(1) time

This is called "direct addressing"

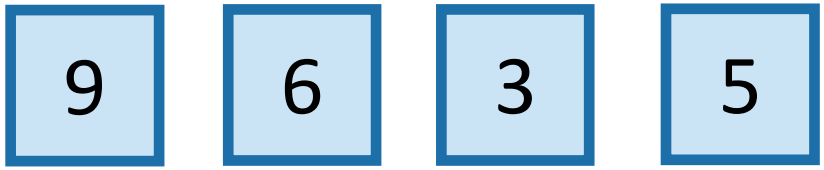
- Say all keys are in the set {1,2,3,4,5,6,7,8,9}.
- INSERT: 9 6 3 5
- DELETE:



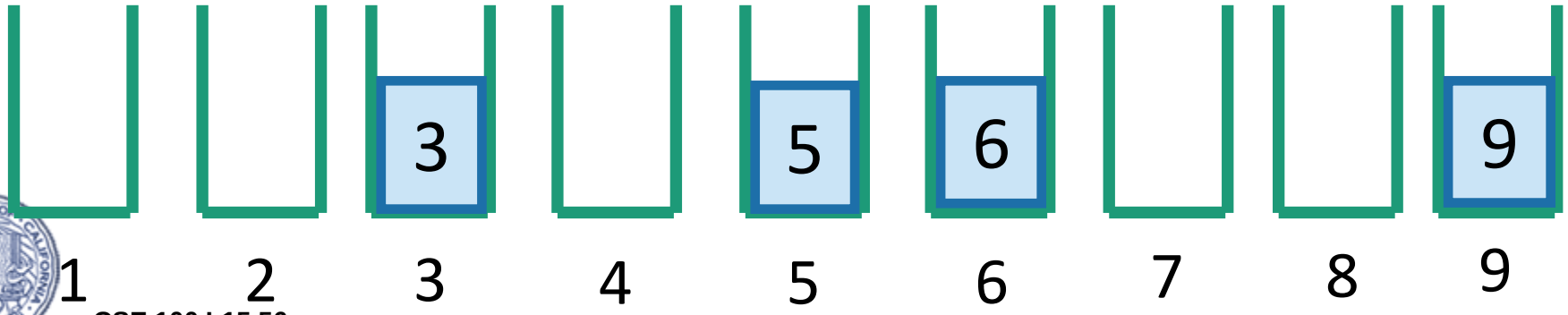
# One way to get O(1) time

This is called "direct addressing"

- Say all keys are in the set {1,2,3,4,5,6,7,8,9}.
- INSERT:



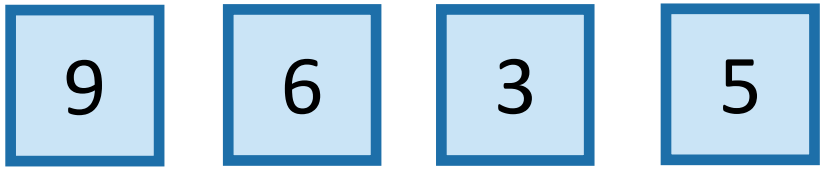

- DELETE:

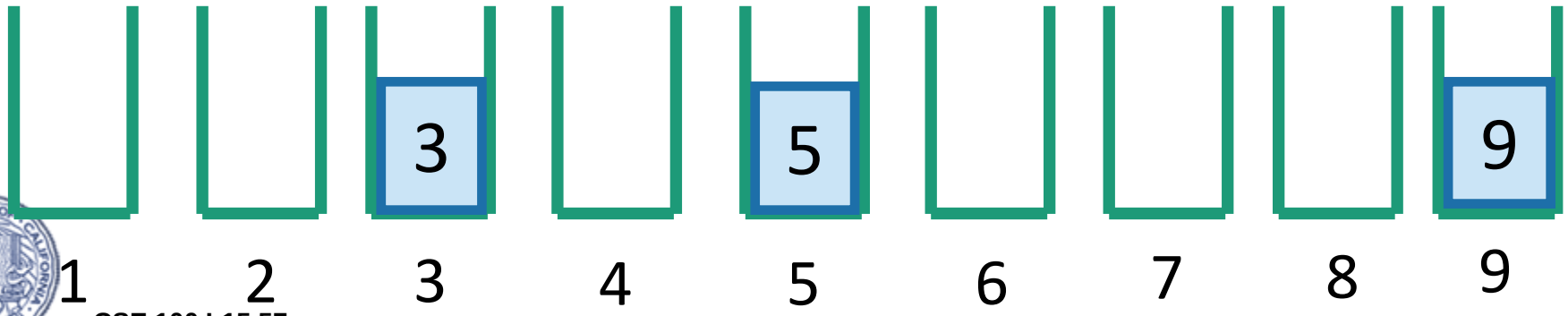
# One way to get O(1) time

This is called "direct addressing"

- Say all keys are in the set {1,2,3,4,5,6,7,8,9}.
- INSERT:



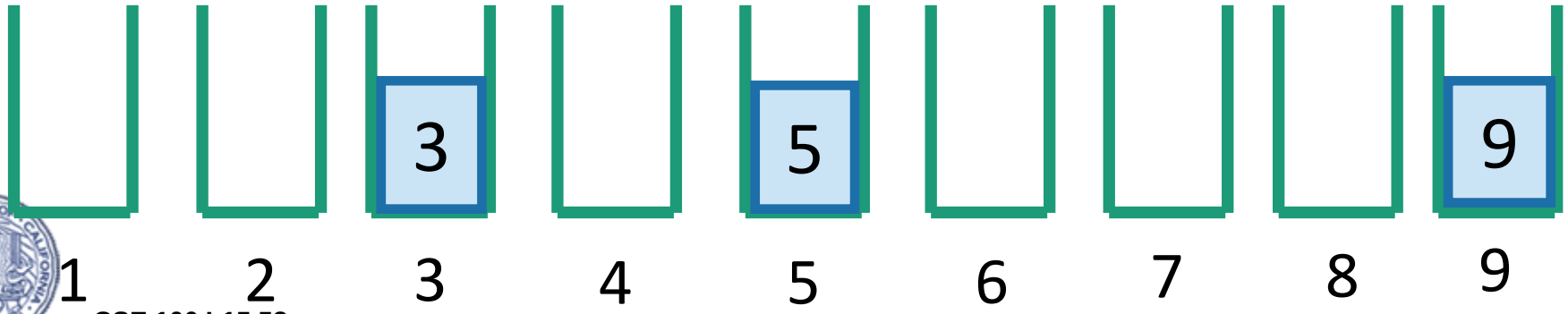

- DELETE:

# One way to get O(1) time

This is called "direct addressing"

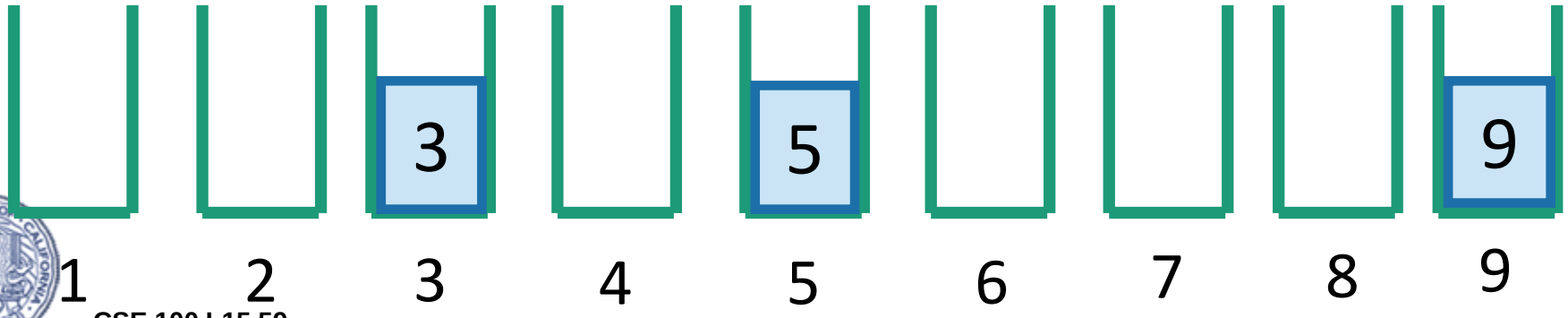
- Say all keys are in the set {1,2,3,4,5,6,7,8,9}.
- INSERT: 9 6 3 5
- DELETE: 6
- SEARCH:



# One way to get O(1) time

This is called "direct addressing"

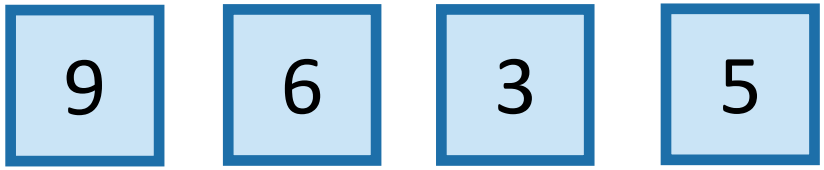
- Say all keys are in the set {1,2,3,4,5,6,7,8,9}.
- INSERT: 9 6 3 5
- DELETE: 6
- SEARCH: 3 2



# One way to get O(1) time

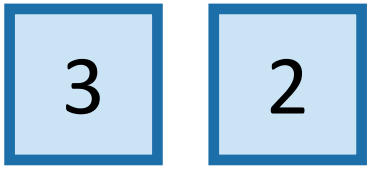
This is called "direct addressing"

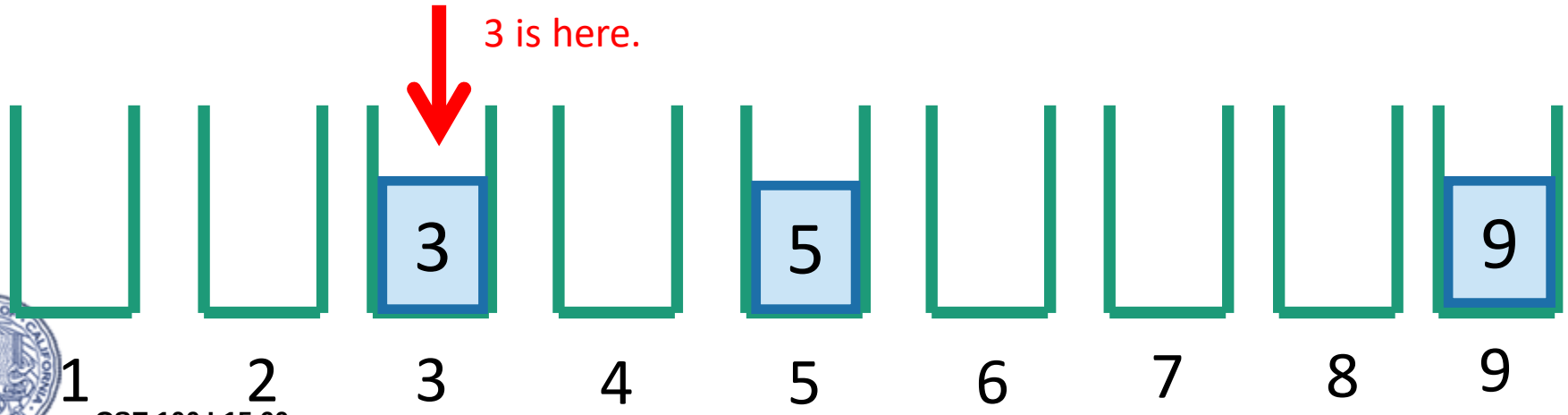
- Say all keys are in the set {1,2,3,4,5,6,7,8,9}.
- INSERT:




- DELETE:


- SEARCH:

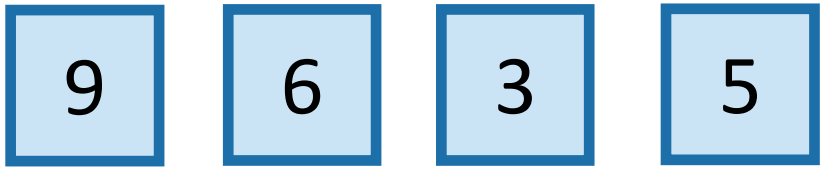


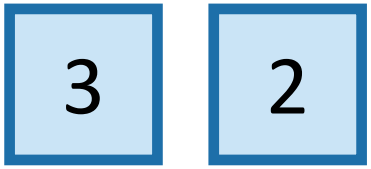
# One way to get O(1) time

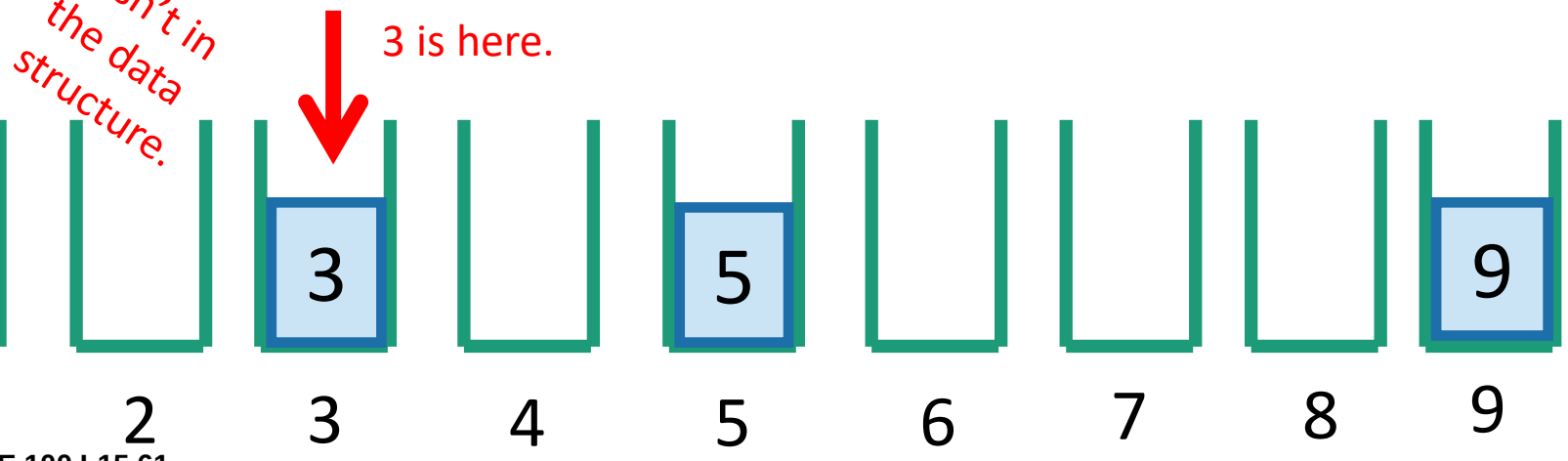
This is called "direct addressing"

- Say all keys are in the set {1,2,3,4,5,6,7,8,9}.
- INSERT:
- DELETE:
- SEARCH:





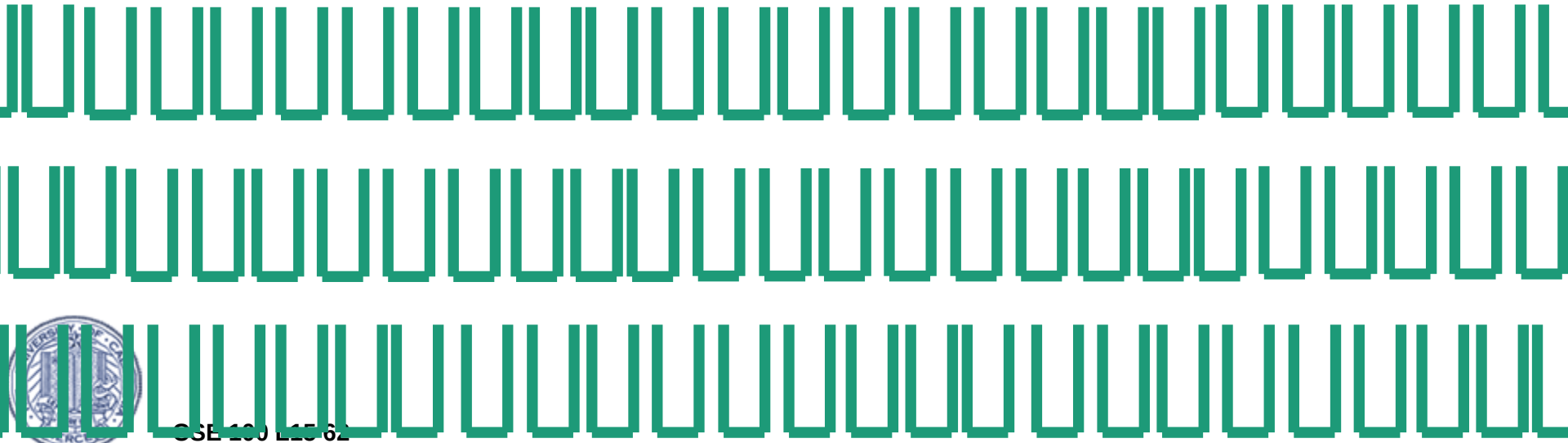


# That should look familiar

The universe is really big!

- Kind of like BUCKETSORT from Lecture 12.
- Same problem: if the keys may come from a "universe" U = {1,2, …., 10000000000}….

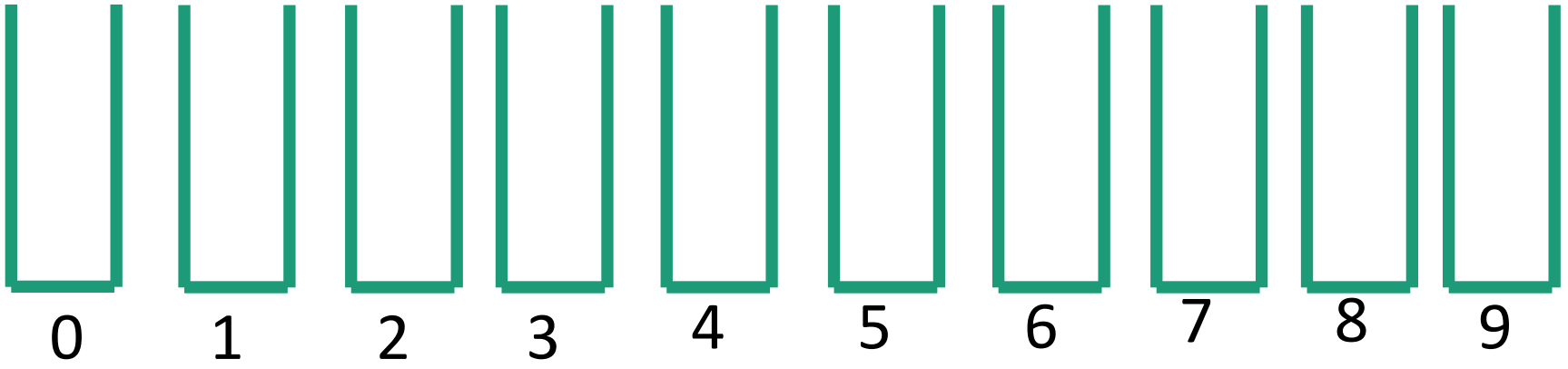


# Solution?

Put things in buckets based on one digit

# Solution?

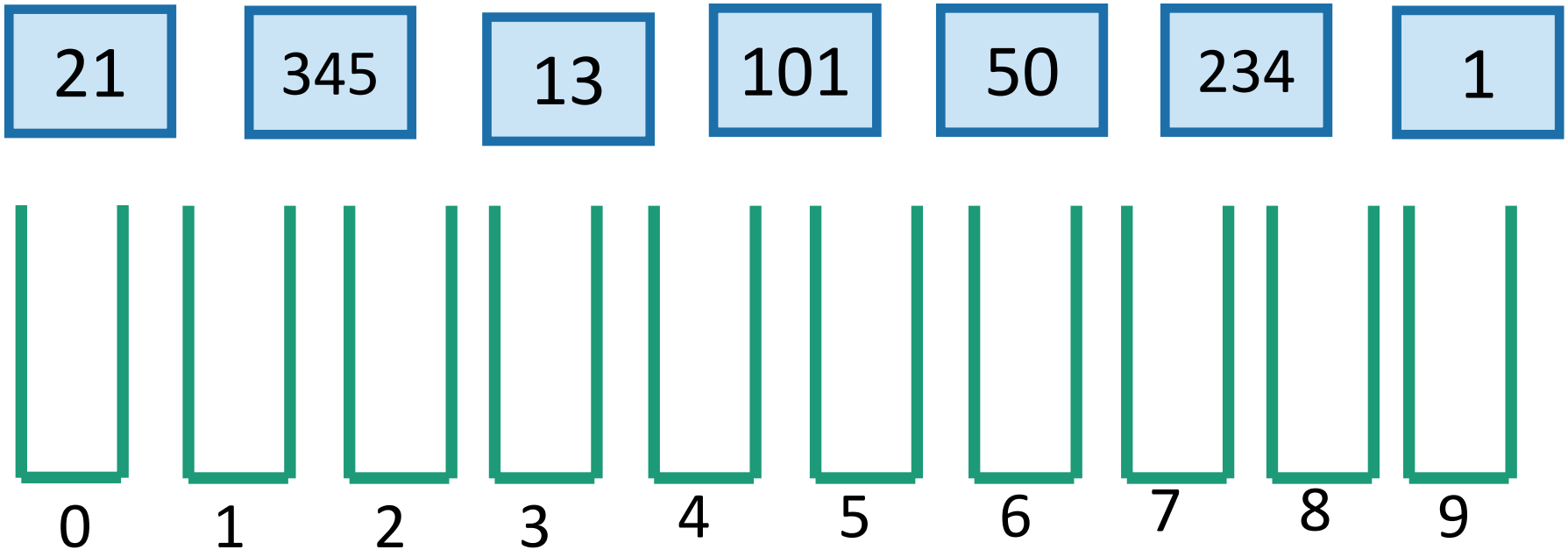
Put things in buckets based on one digit




# Solution?

Put things in buckets based on one digit

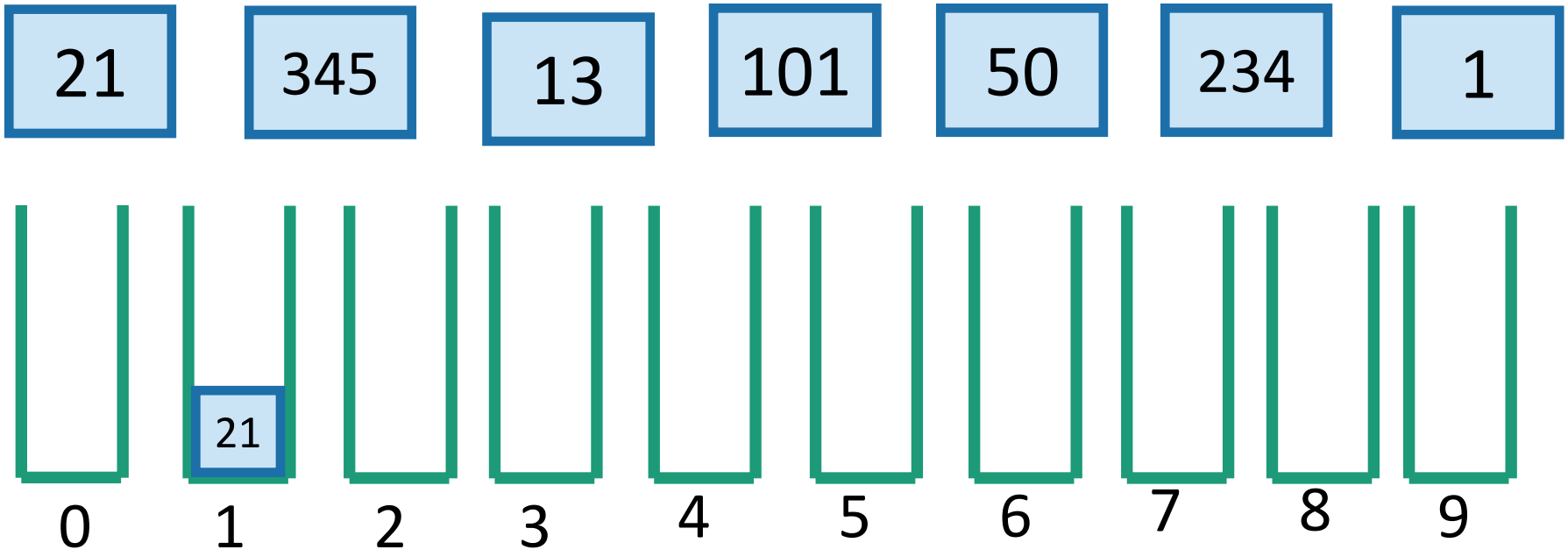
INSERT:




# Solution?

Put things in buckets based on one digit

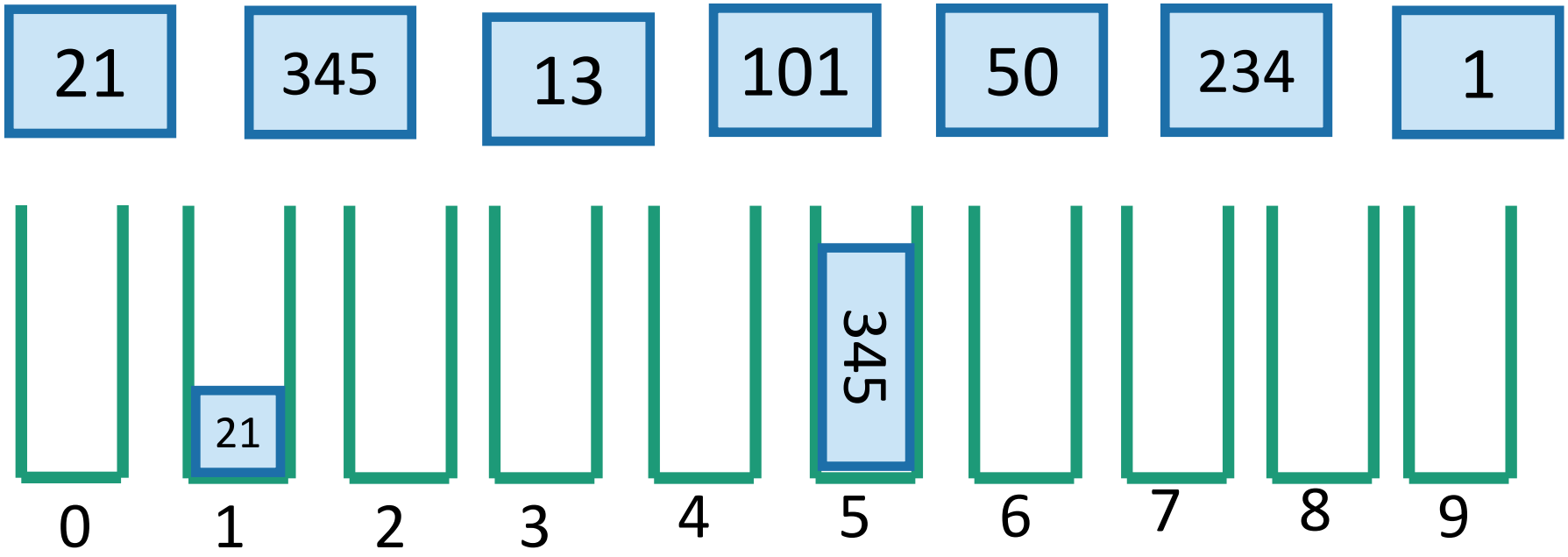
INSERT:




# Solution?

Put things in buckets based on one digit

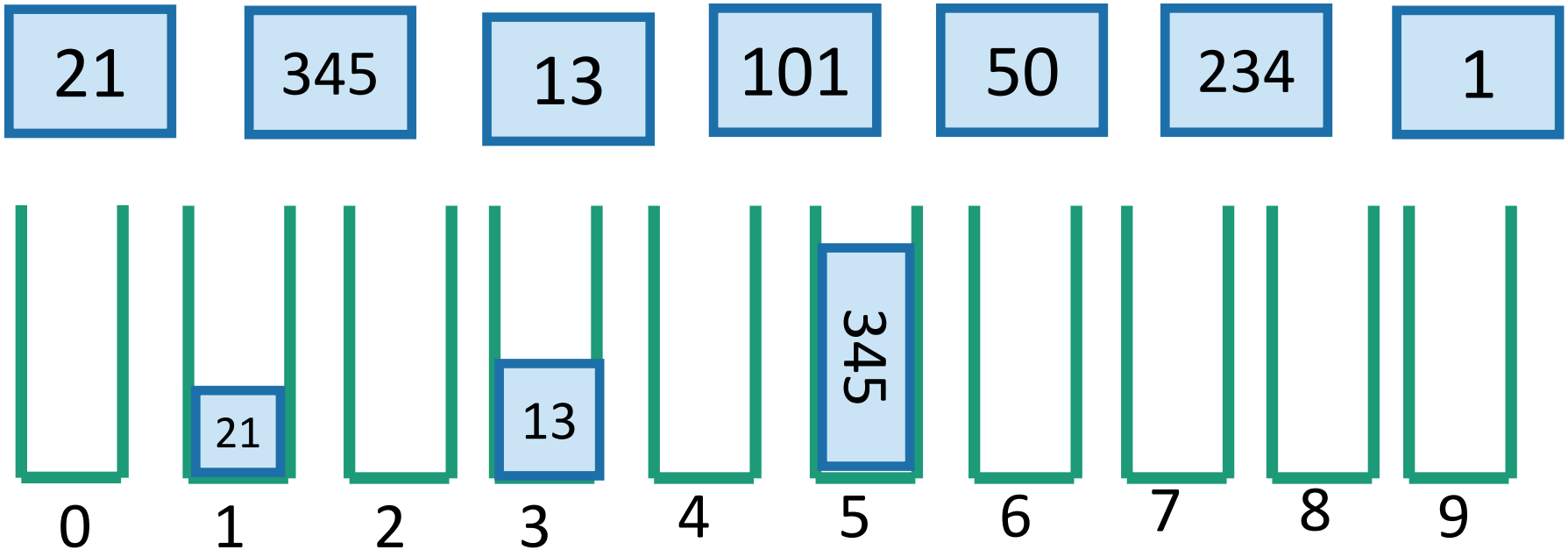
INSERT:




# Solution?

Put things in buckets based on one digit

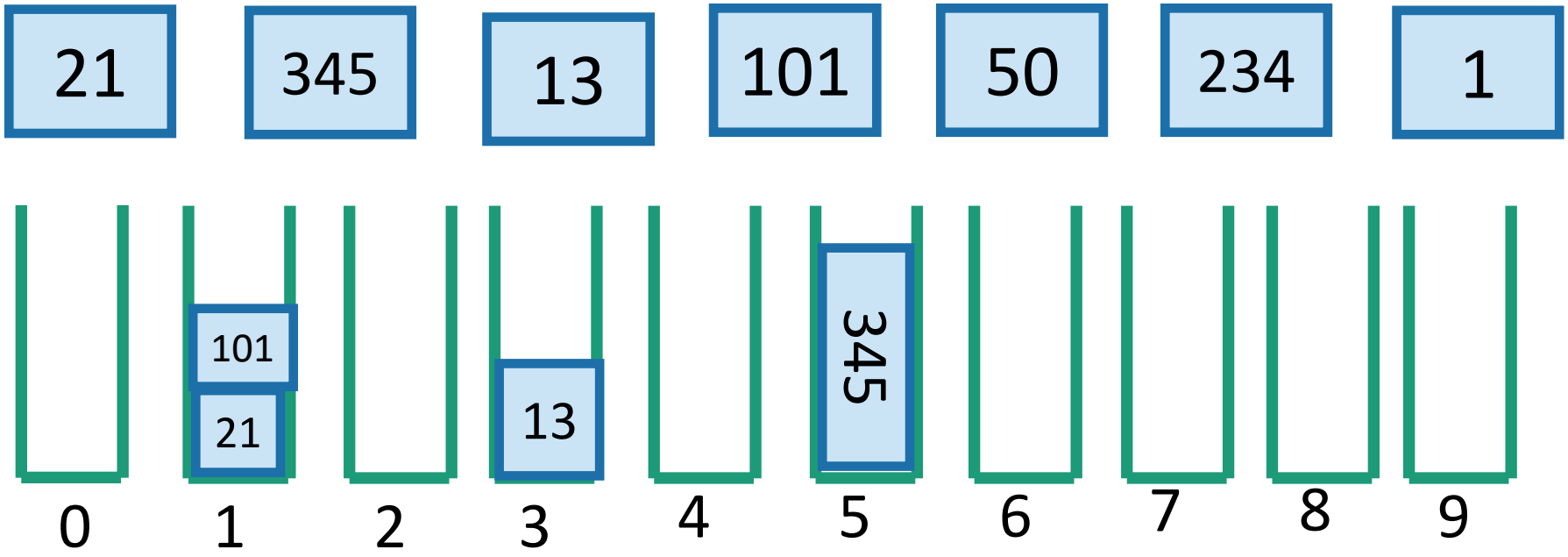
INSERT:




# Solution?

Put things in buckets based on one digit

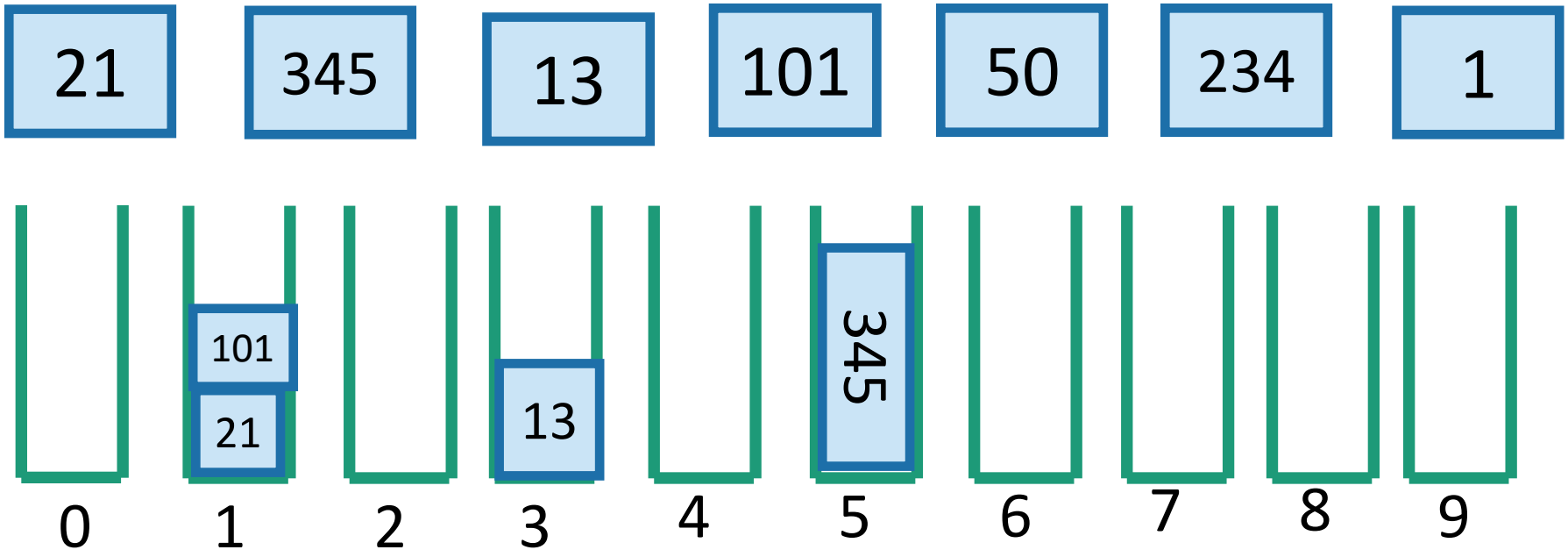
INSERT:




# Solution?

Put things in buckets based on one digit

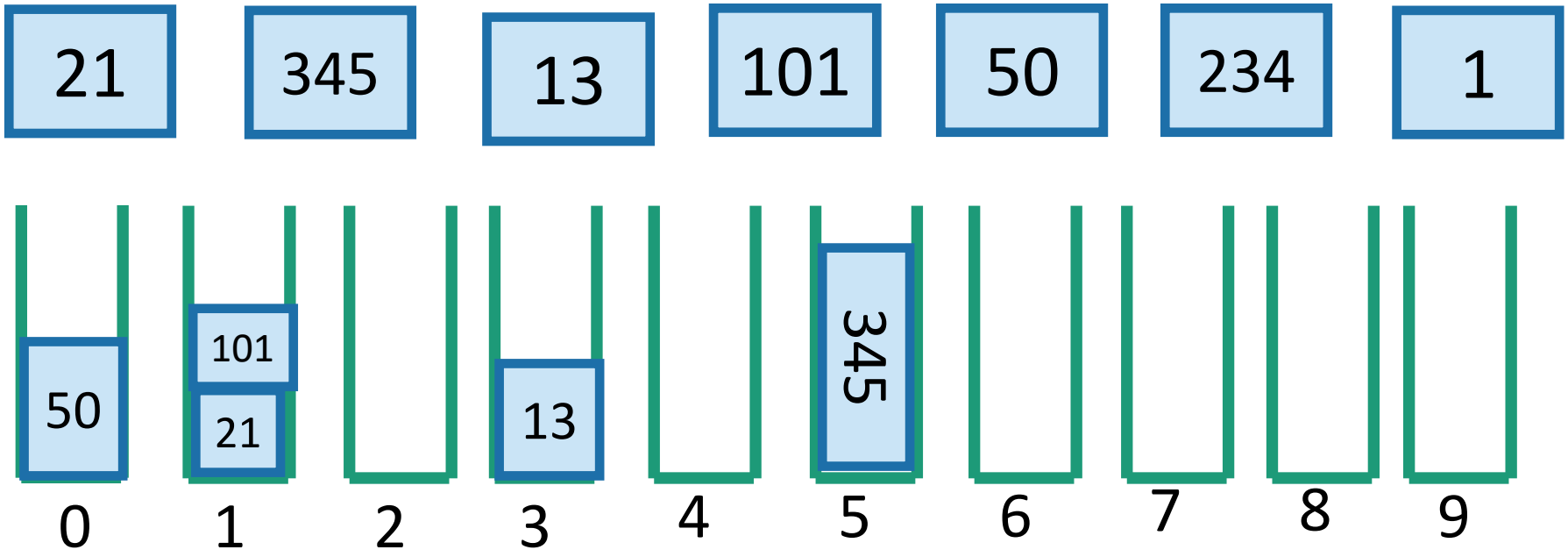
INSERT:




# Solution?

Put things in buckets based on one digit

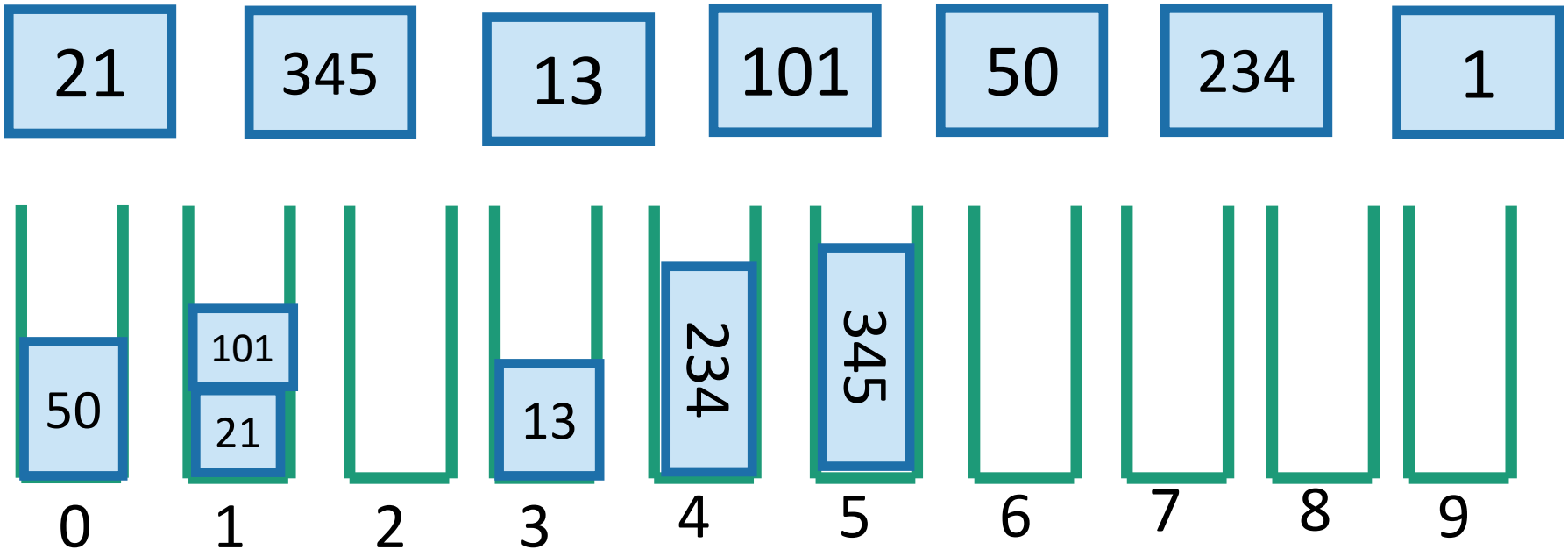
INSERT:




# Solution?

Put things in buckets based on one digit

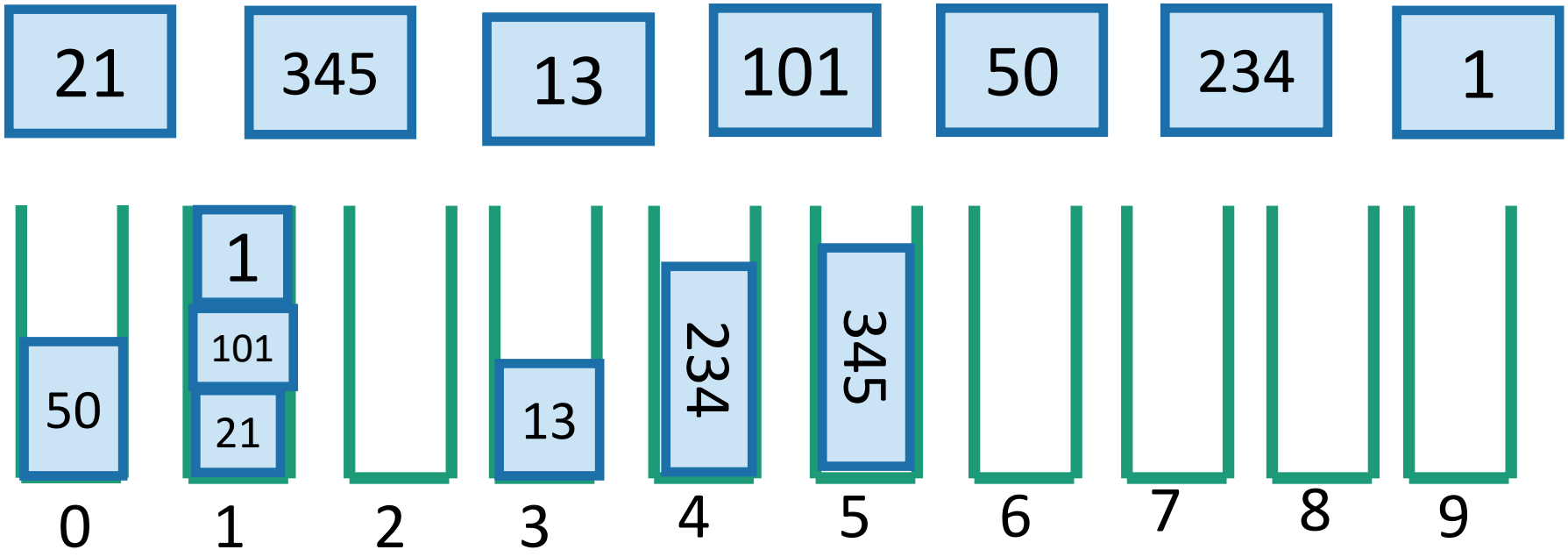
INSERT:




# Solution?

Put things in buckets based on one digit

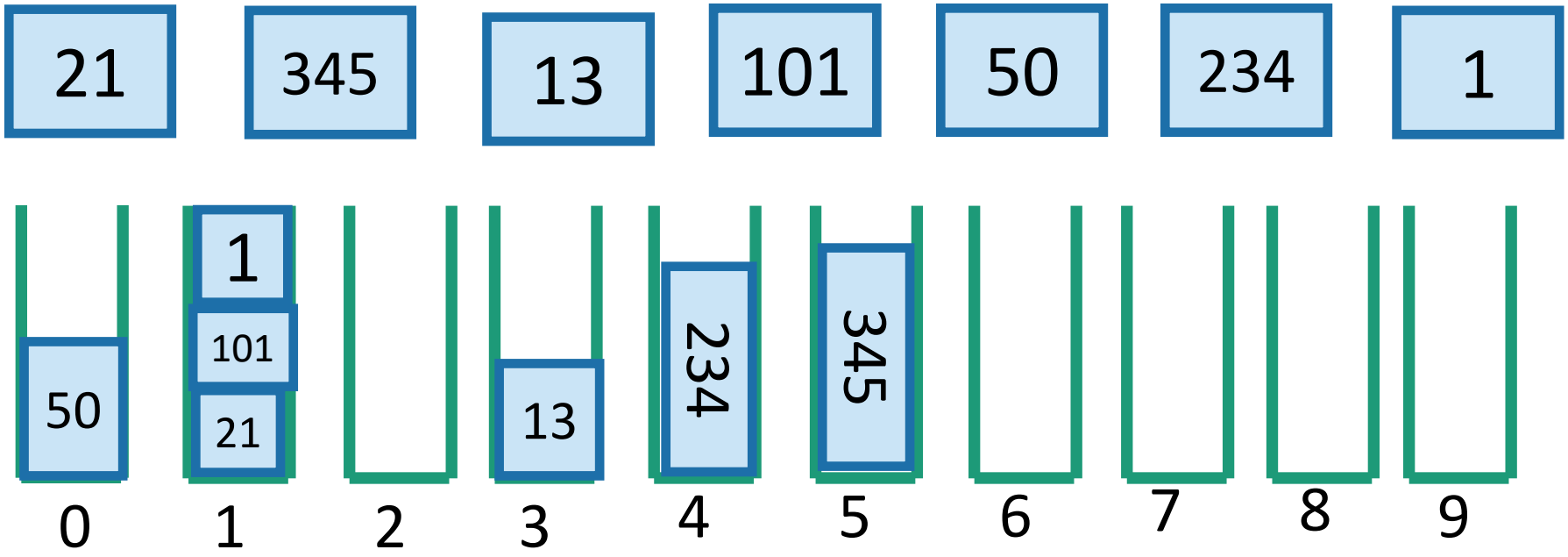
INSERT:



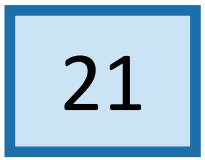

# Solution?

Put things in buckets based on one digit

INSERT:



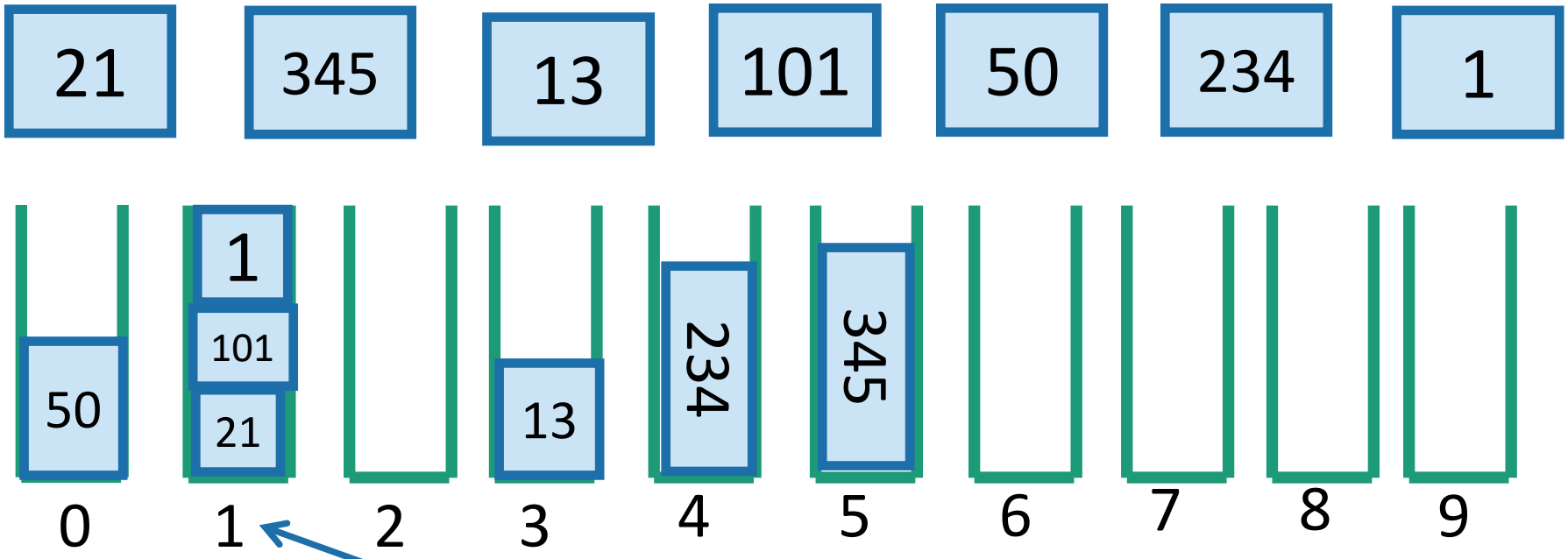

Now SEARCH



# Solution?

Put things in buckets based on one digit

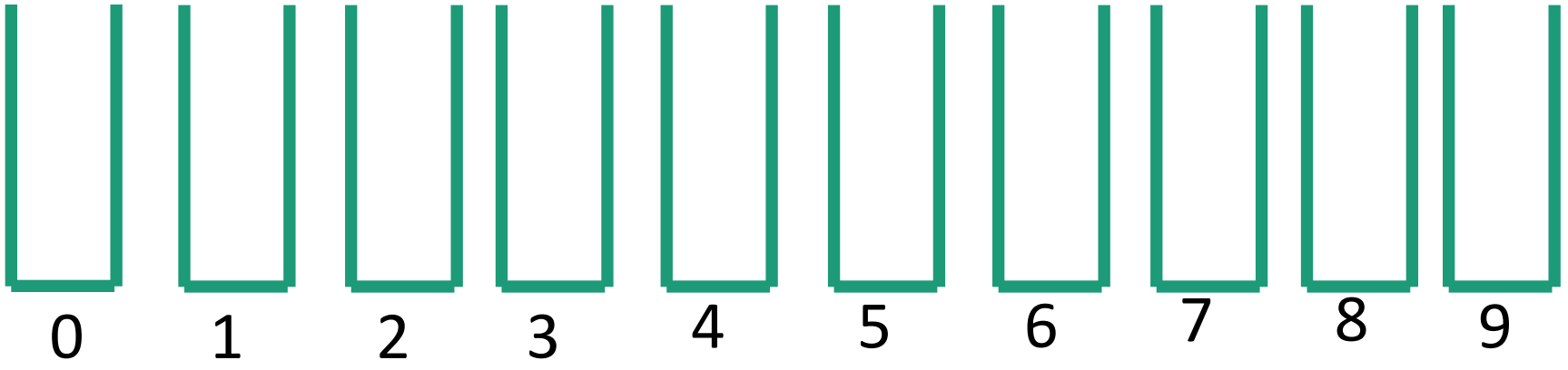
INSERT:




It's in this bucket somewhere…
go through until we find it.

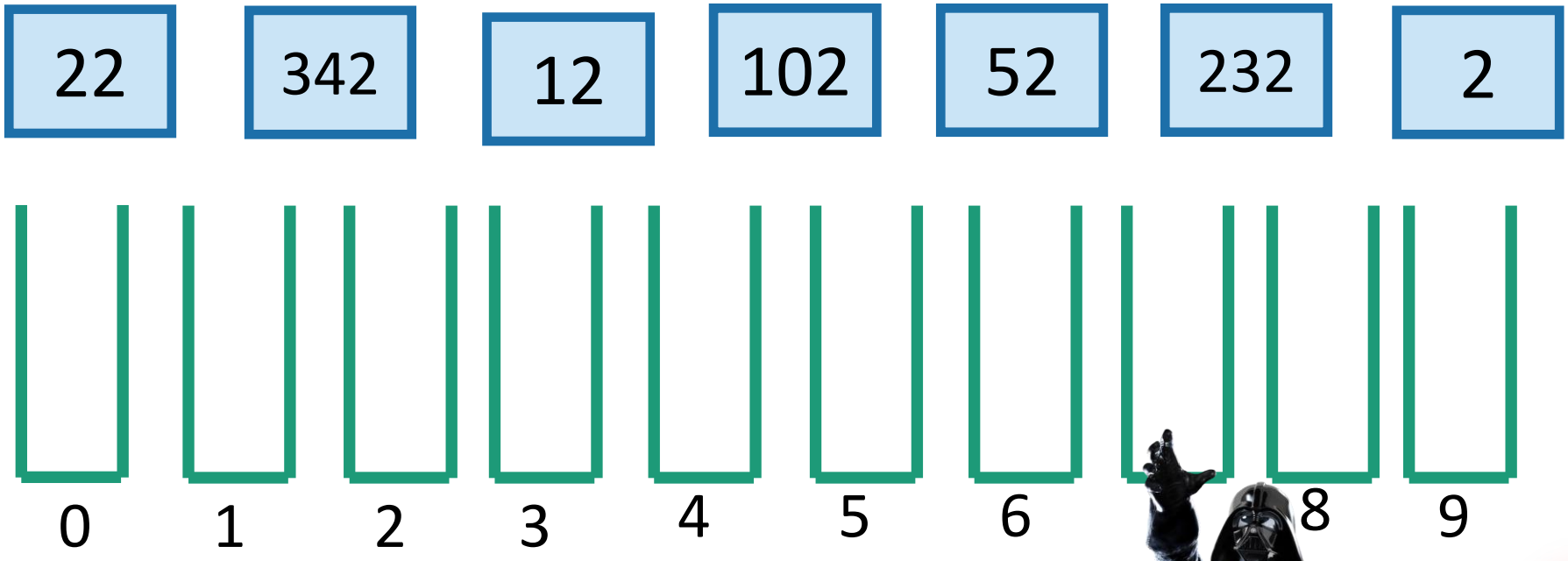
Now SEARCH 21

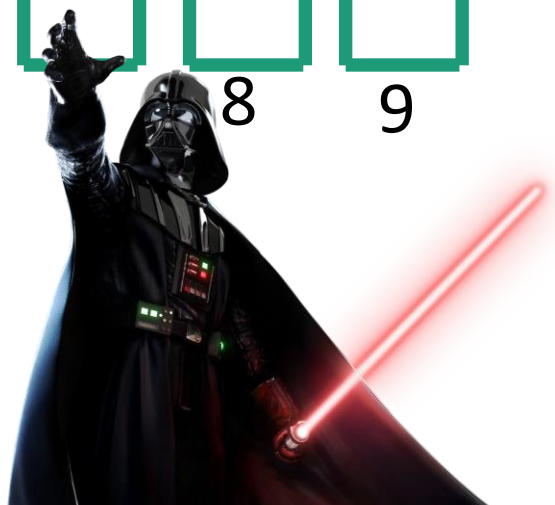
# Problem…




# Problem…

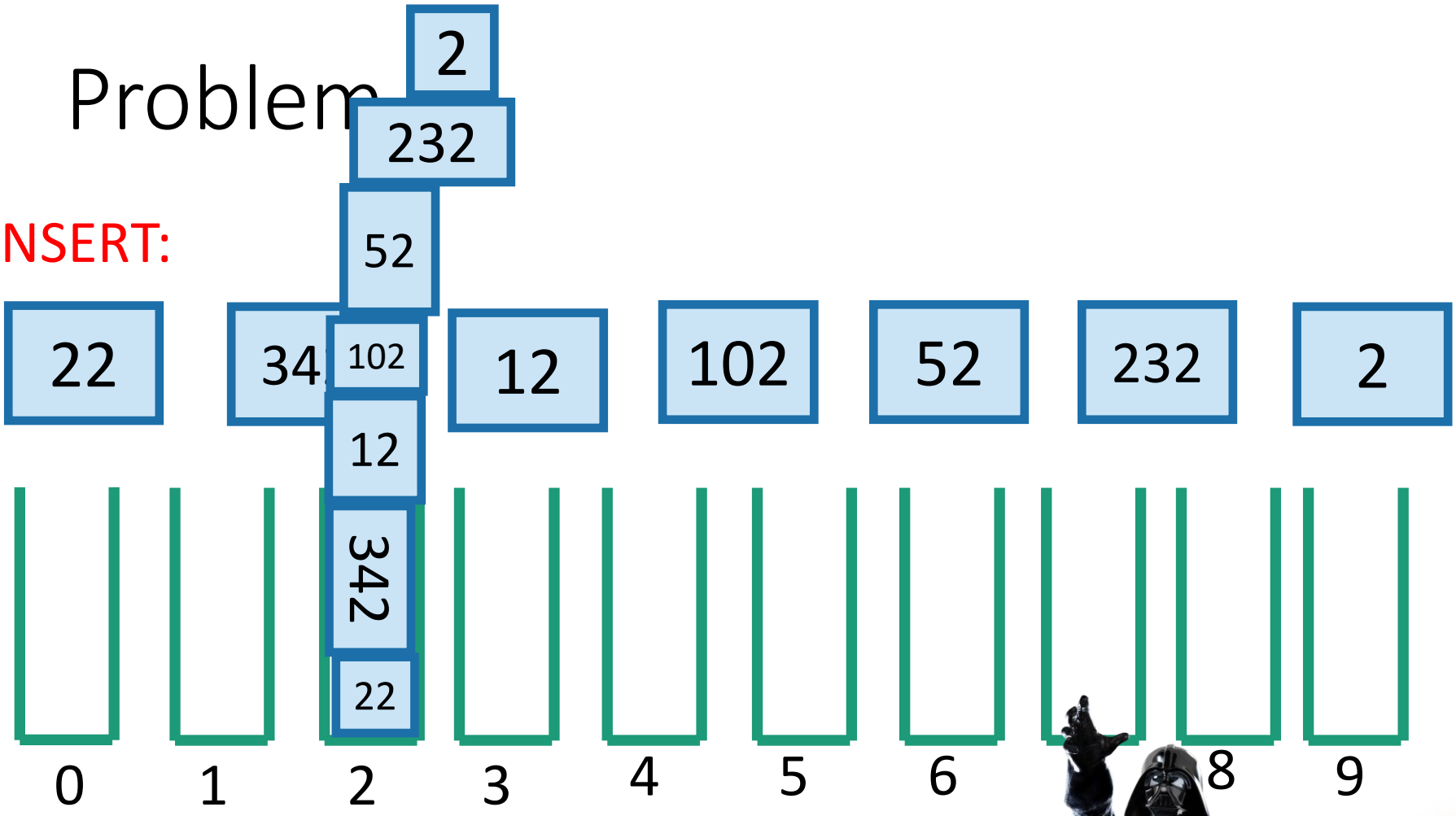
INSERT:

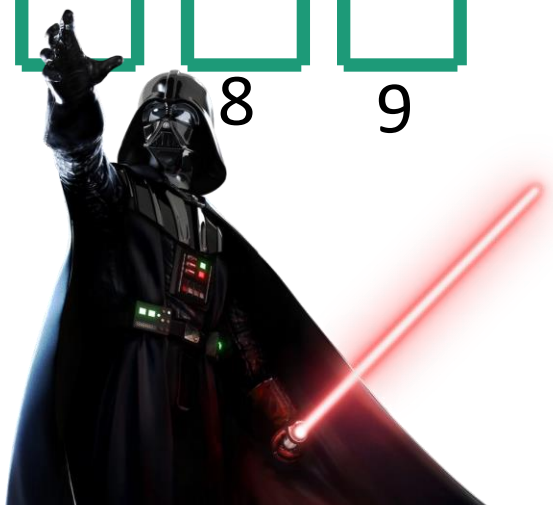


# Problem

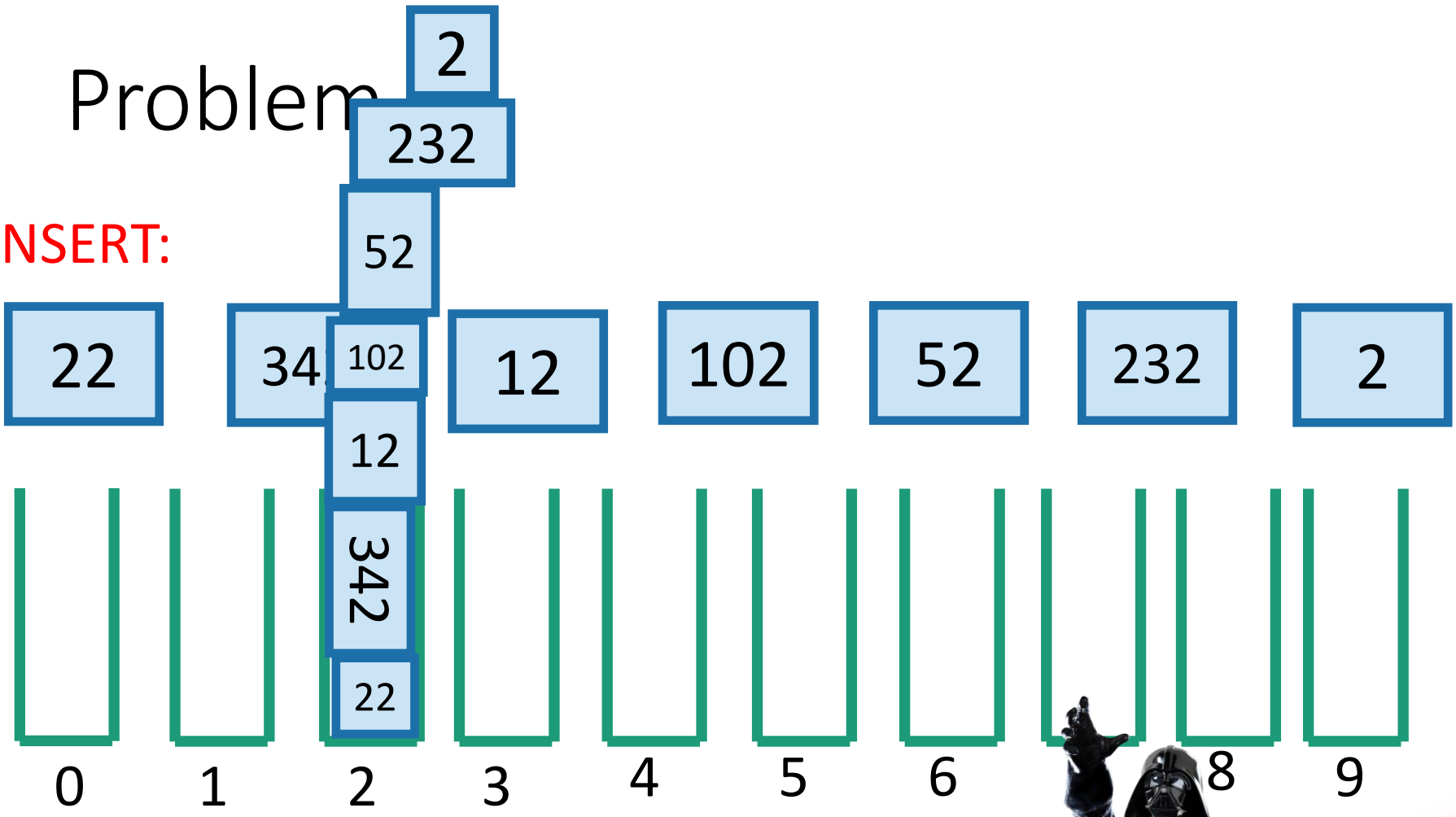
INSERT:



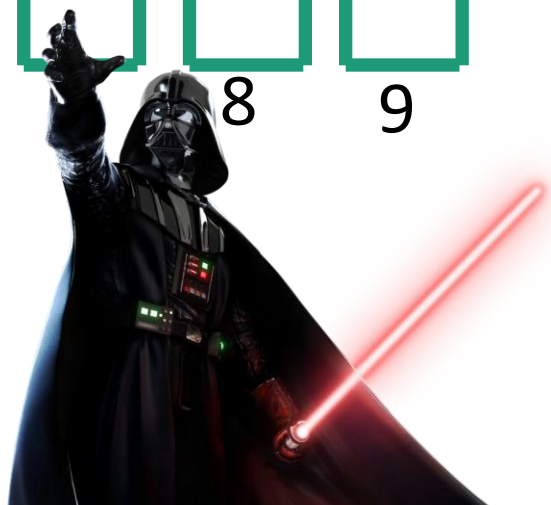



# Problem

INSERT:

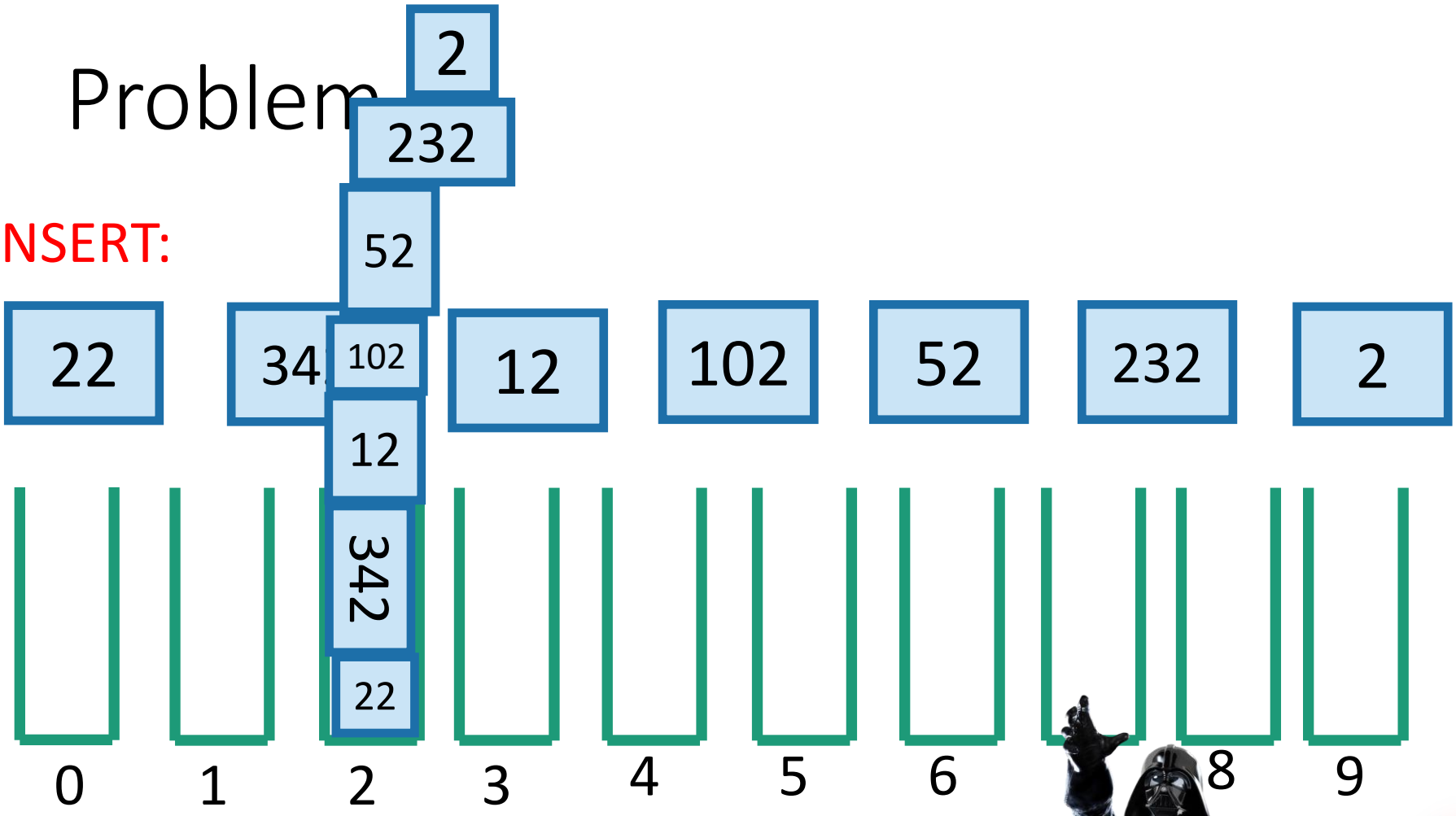



Now SEARCH 22



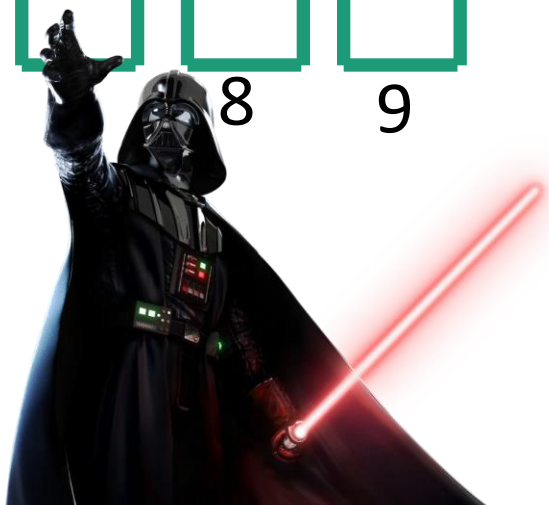
# Problem

INSERT:




Now SEARCH 22

….this hasn't made our lives easier…

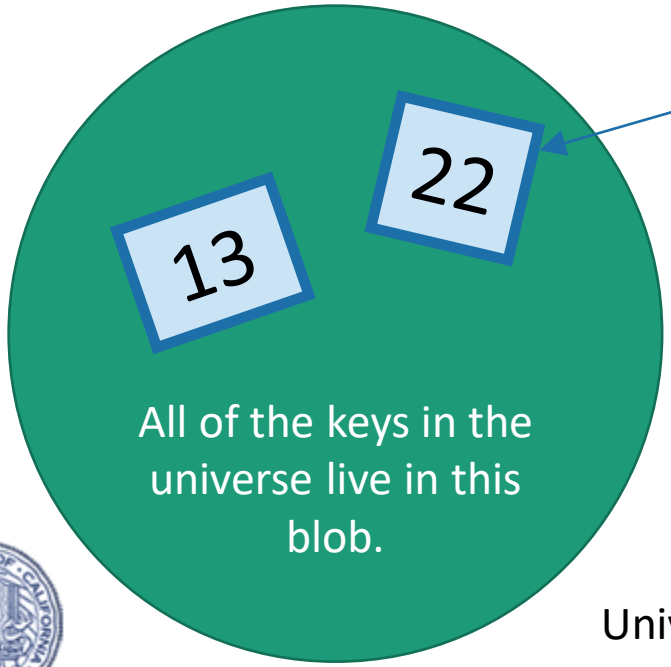


# Hash tables

- That was an example of a hash table.
  - not a very good one, though.
- We will be **more clever** (and less deterministic) about our bucketing.
- This will result in fast (expected time) INSERT/DELETE/SEARCH.

# But first! Terminology.

- We have a universe U, of size M.
  - M is really big.
- But only a few (say at most n for today's lecture) elements of M are ever going to show up.
  - M is waaaayyyyyyy bigger than n.
- But we don't know which ones will show up in advance.




A few elements are special and will actually show up.

Universe U

Example: U is the set of all strings of at most 140 ascii characters. ($256^{280}$ of them).

The only ones which I care about are those which appear as trending hashtags on twitter. #hashinghashtags

*There are way fewer than $256^{280}$ of these.*

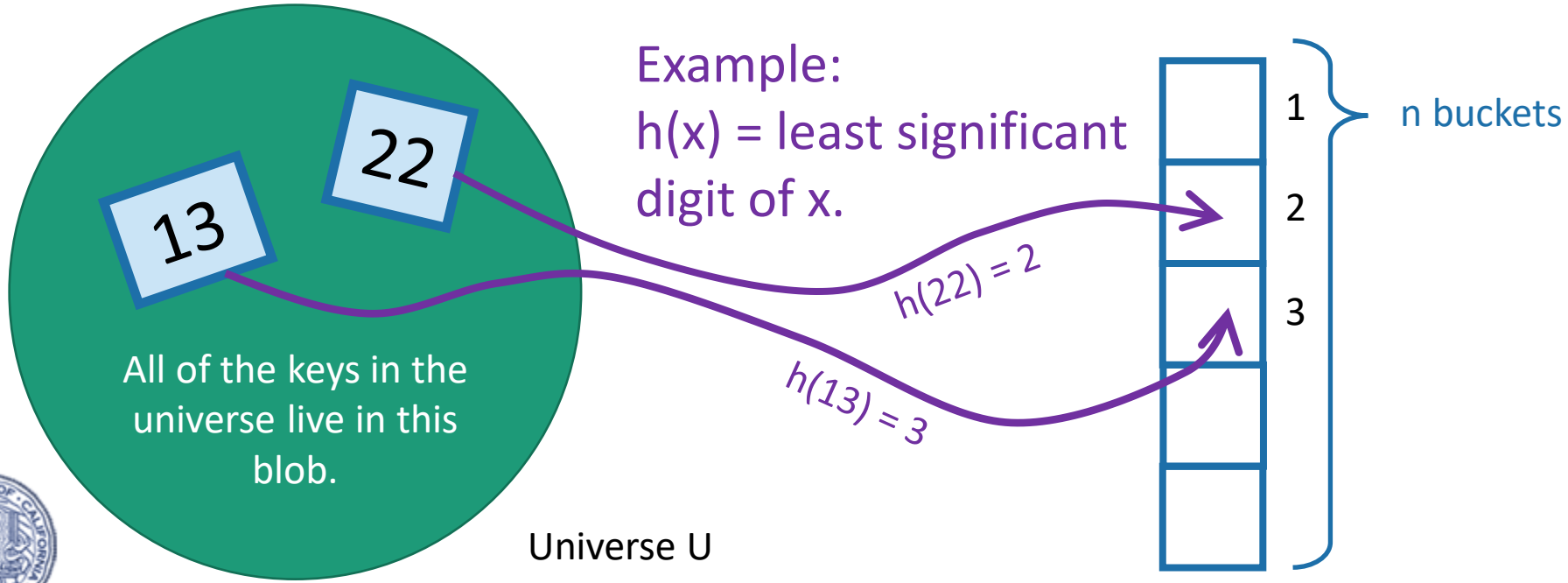
Examples aside, I'm going to draw elements like I always do, as blue boxes with integers in them…

# The previous example

with this terminology

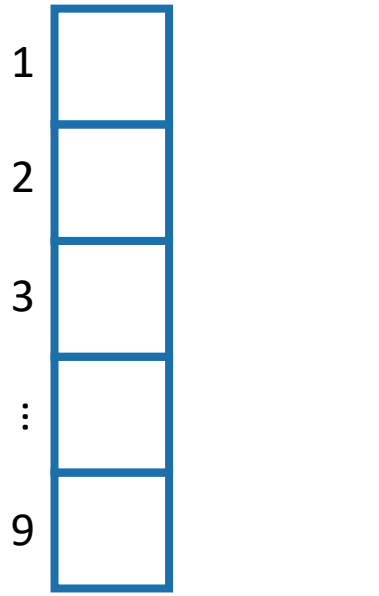
For this lecture, I'm assuming that the number of things is the same as the number of buckets, both are n.
This doesn't have to be the case, although we do want:
**#buckets = O( #things which show up )**

- We have a universe U, of size M.
  - at most n of which will show up.
- M is waaaayyyyyy bigger than n.
- We will put items of U into n buckets.
- There is a *hash function* $h: U \to \{1, \dots, n\}$ which says what element goes in what bucket.



# This is a **hash table** (with chaining)

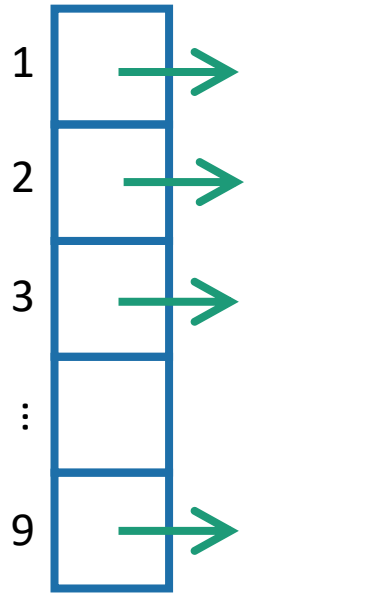
- Array of n buckets.



n buckets (say n=9)

# This is a **hash table** (with chaining)

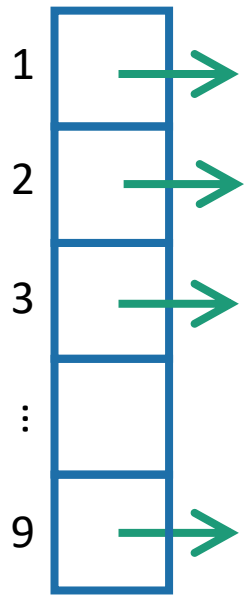
- Array of n buckets.
- Each bucket stores a linked list.
  - We can insert into a linked list in time O(1)
  - To find something in the linked list takes time O(length(list)).



n buckets (say n=9)

# This is a **hash table** (with chaining)

- Array of n buckets.
- Each bucket stores a linked list.
  - We can insert into a linked list in time O(1)
  - To find something in the linked list takes time O(length(list)).
- $h: U \to \{1, \ldots, n\}$ can be any function:
  - but for concreteness let's stick with h(x) = least significant digit of x.

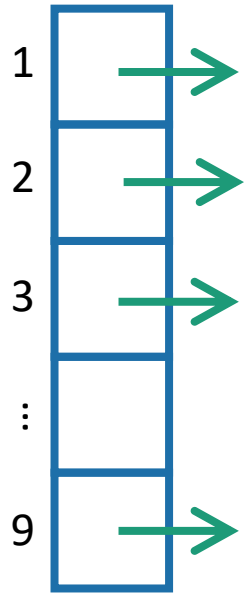


n buckets (say n=9)

# This is a **hash table** (with chaining)

- Array of n buckets.
- Each bucket stores a linked list.
  - We can insert into a linked list in time O(1)
  - To find something in the linked list takes time O(length(list)).
- $h: U \to \{1, \ldots, n\}$ can be any function:
  - but for concreteness let's stick with h(x) = least significant digit of x.

**For demonstration purposes only!**
This is a terrible hash function! Don't use this!

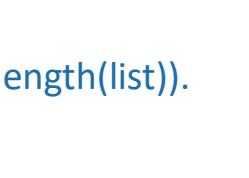


n buckets (say n=9)

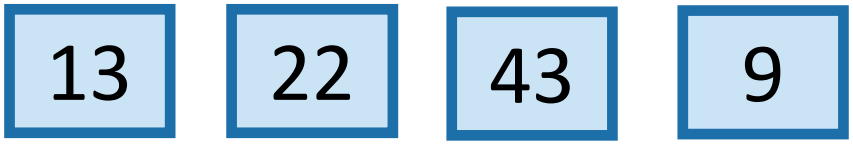
# This is a **hash table** (with chaining)

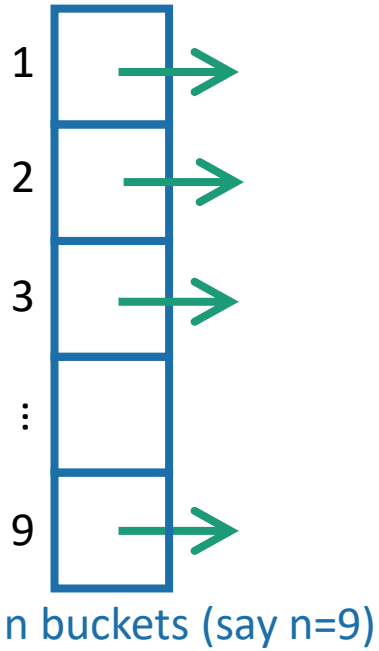
- Array of n buckets.
- Each bucket stores a linked list.
  - We can insert into a linked list in time O(1)
  - To find something in the linked list takes time O(length(list)).
- $h: U \to \{1, \ldots, n\}$ can be any function:
  - but for concreteness let's stick with h(x) = least significant digit of x.

**For demonstration purposes only!**
This is a terrible hash function! Don't use this!



INSERT:



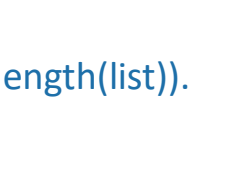




n buckets (say n=9)

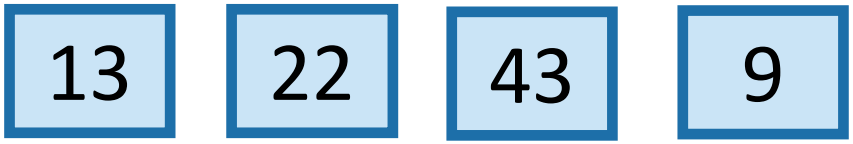
# This is a **hash table** (with chaining)

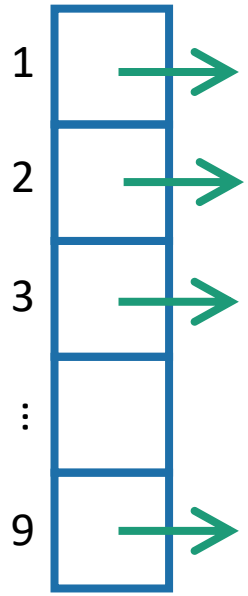
- Array of n buckets.
- Each bucket stores a linked list.
  - We can insert into a linked list in time O(1)
  - To find something in the linked list takes time O(length(list)).
- $h: U \to \{1, \ldots, n\}$ can be any function:
  - but for concreteness let's stick with h(x) = least significant digit of x.

**For demonstration purposes only!**
This is a terrible hash function! Don't use this!



INSERT:



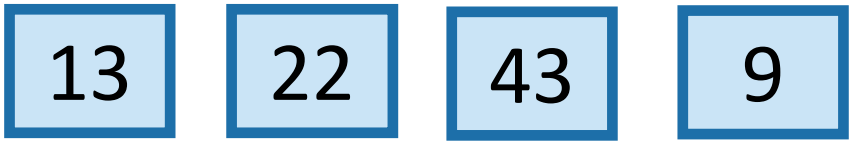

n buckets (say n=9)

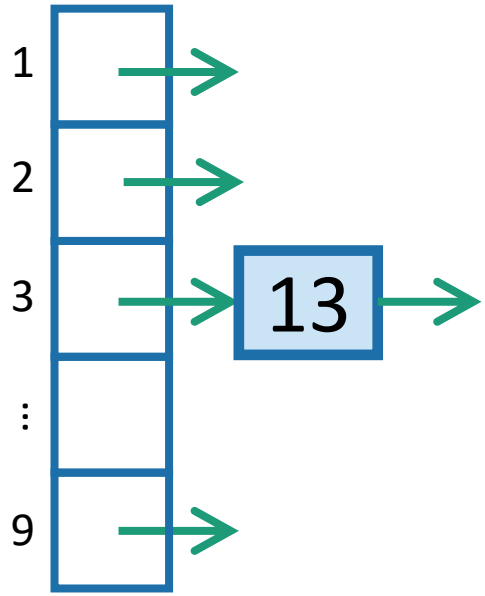
# This is a **hash table** (with chaining)

- Array of n buckets.
- Each bucket stores a linked list.
  - We can insert into a linked list in time O(1)
  - To find something in the linked list takes time O(length(list)).
- $h: U \to \{1, \ldots, n\}$ can be any function:
  - but for concreteness let's stick with h(x) = least significant digit of x.

**For demonstration purposes only!**
This is a terrible hash function! Don't use this!

INSERT:



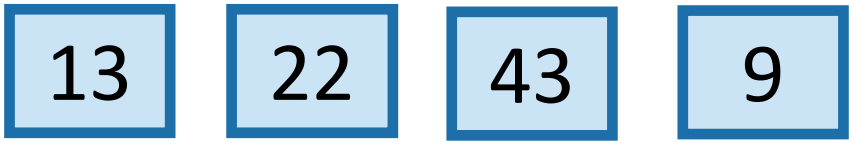

n buckets (say n=9)

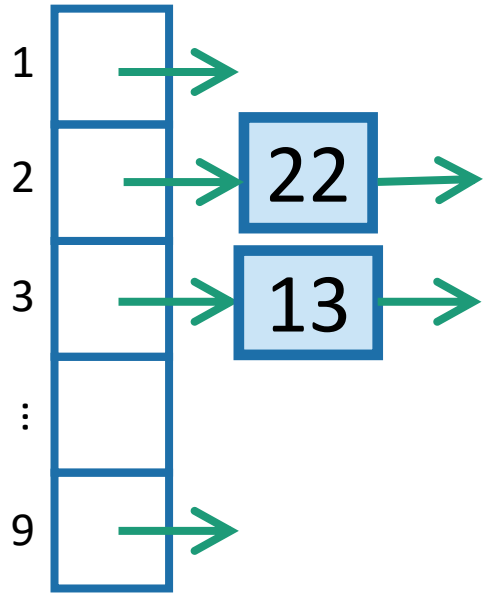
# This is a **hash table** (with chaining)

- Array of n buckets.
- Each bucket stores a linked list.
  - We can insert into a linked list in time O(1)
  - To find something in the linked list takes time O(length(list)).
- $h: U \to \{1, \ldots, n\}$ can be any function:
  - but for concreteness let's stick with h(x) = least significant digit of x.

**For demonstration purposes only!**
This is a terrible hash function! Don't use this!

INSERT:



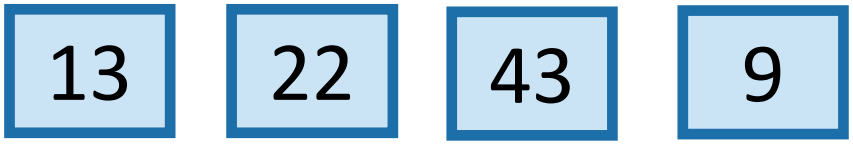

n buckets (say n=9)

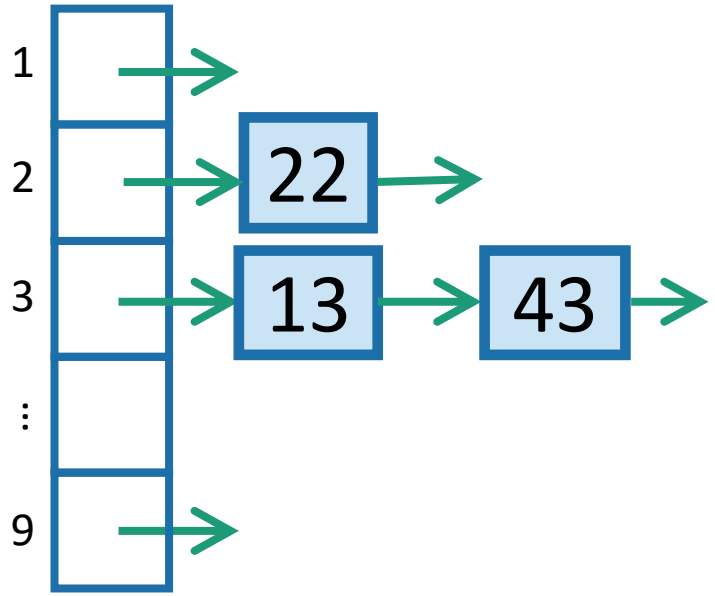
# This is a **hash table** (with chaining)

- Array of n buckets.
- Each bucket stores a linked list.
  - We can insert into a linked list in time O(1)
  - To find something in the linked list takes time O(length(list)).
- $h: U \to \{1, \ldots, n\}$ can be any function:
  - but for concreteness let's stick with h(x) = least significant digit of x.

**For demonstration purposes only!**
This is a terrible hash function! Don't use this!

INSERT:



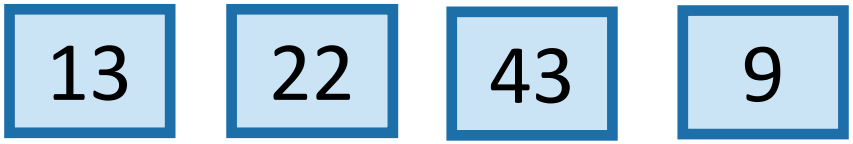

n buckets (say n=9)

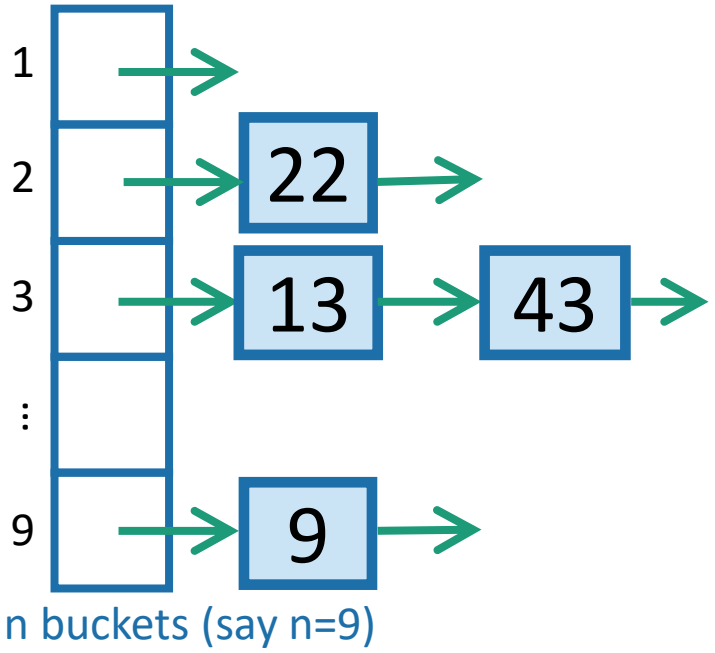
# This is a **hash table** (with chaining)

- Array of n buckets.
- Each bucket stores a linked list.
  - We can insert into a linked list in time O(1)
  - To find something in the linked list takes time O(length(list)).
- $h: U \to \{1, \ldots, n\}$ can be any function:
  - but for concreteness let's stick with h(x) = least significant digit of x.

**For demonstration purposes only!**
This is a terrible hash function! Don't use this!

INSERT:



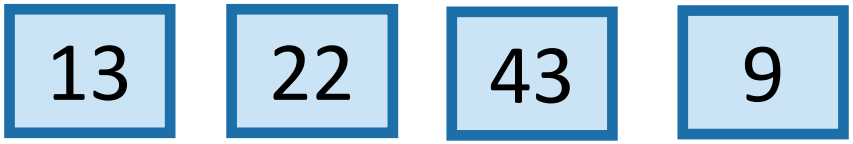
n buckets (say n=9)

# This is a **hash table** (with chaining)

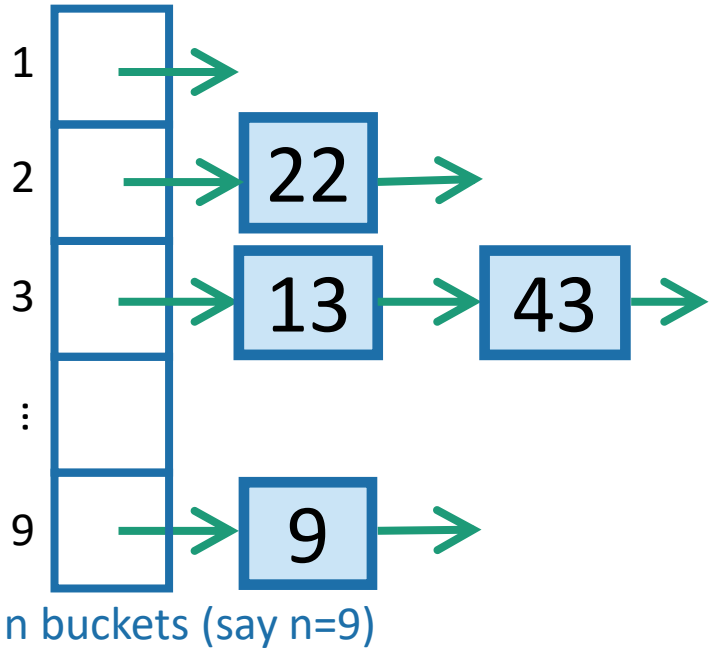
- Array of n buckets.
- Each bucket stores a linked list.
  - We can insert into a linked list in time O(1)
  - To find something in the linked list takes time O(length(list)).
- $h: U \to \{1, \ldots, n\}$ can be any function:
  - but for concreteness let's stick with h(x) = least significant digit of x.

**For demonstration purposes only!**
This is a terrible hash function! Don't use this!

INSERT:




SEARCH 43:



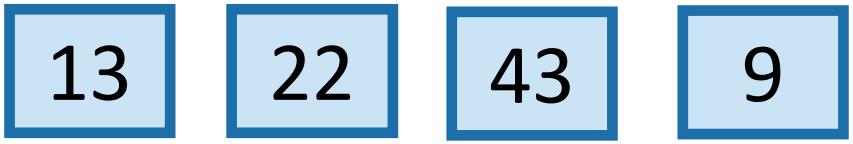

n buckets (say n=9)

# This is a **hash table** (with chaining)

- Array of n buckets.
- Each bucket stores a linked list.
  - We can insert into a linked list in time O(1)
  - To find something in the linked list takes time O(length(list)).
- $h: U \to \{1, \ldots, n\}$ can be any function:
  - but for concreteness let's stick with h(x) = least significant digit of x.

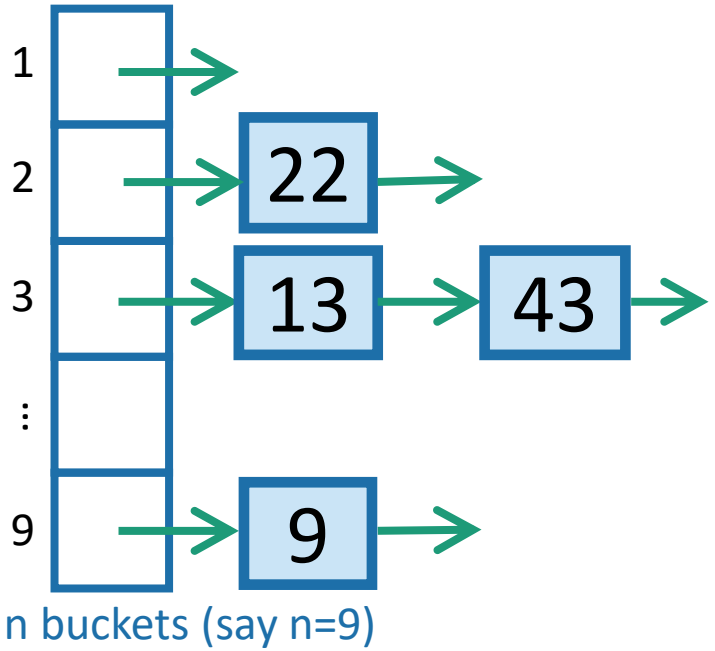
**For demonstration purposes only!**
This is a terrible hash function! Don't use this!

INSERT:




SEARCH 43:

Scan through all the elements in bucket h(43) = 3.




n buckets (say n=9)

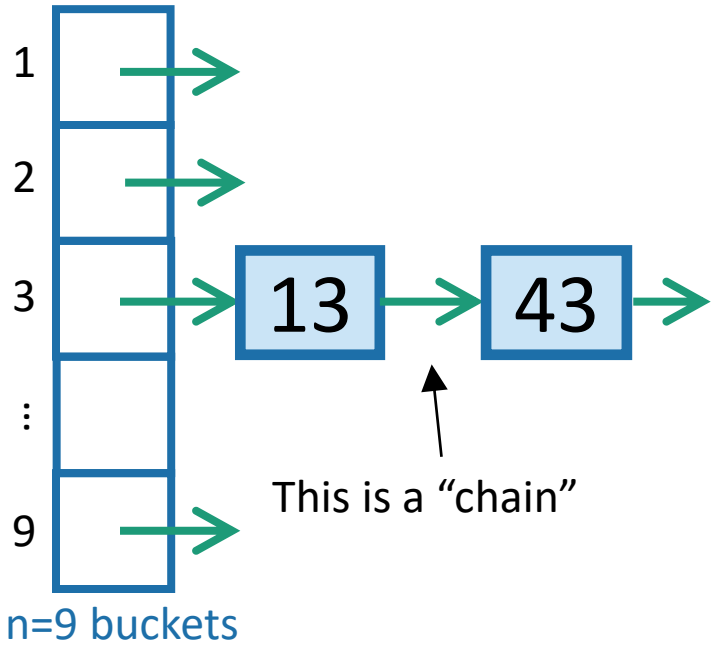
# Aside: Hash tables with open addressing

# Aside: Hash tables with open addressing

- The previous slide is about hash tables with chaining.

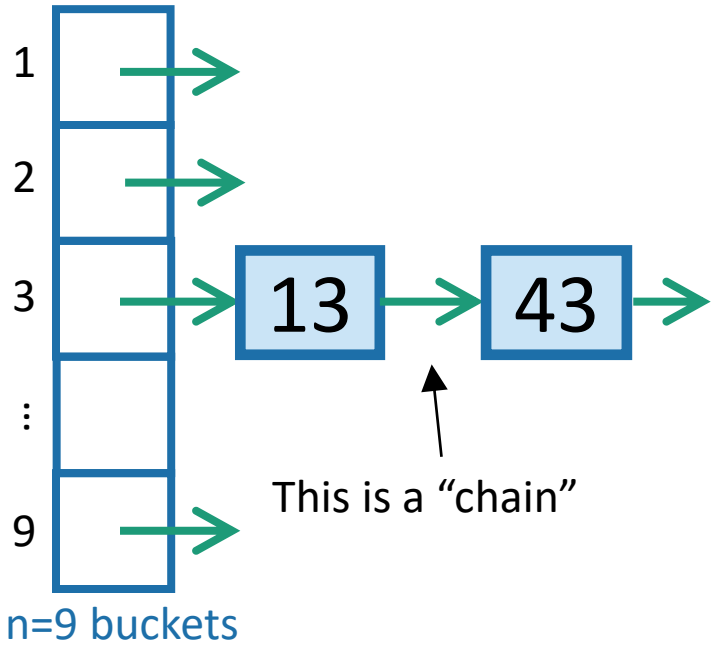
# Aside: Hash tables with open addressing

- The previous slide is about hash tables with chaining.



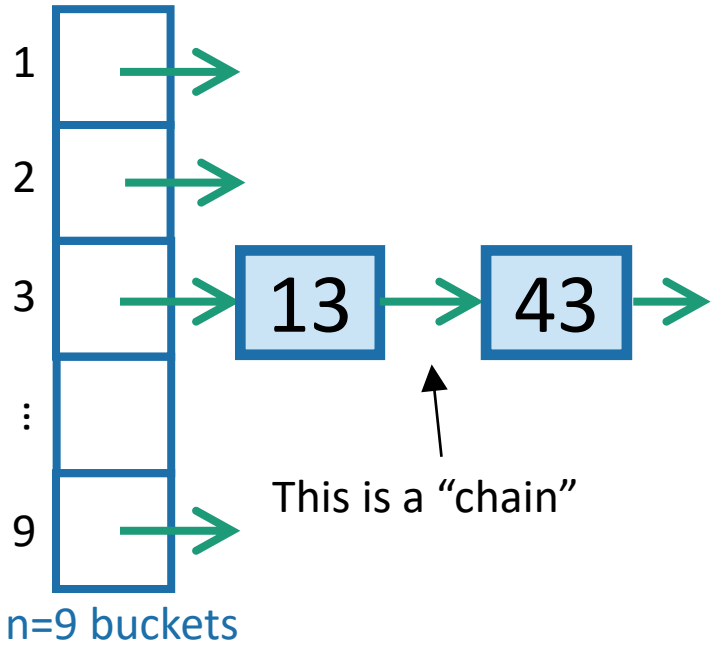

# Aside: Hash tables with open addressing

- The previous slide is about hash tables with chaining.
- There's also something called "open addressing"



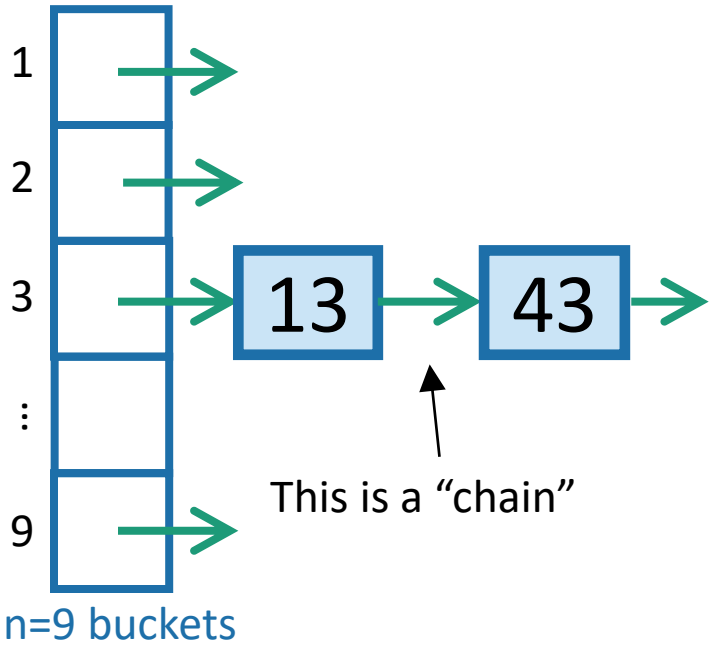

# Aside: Hash tables with open addressing

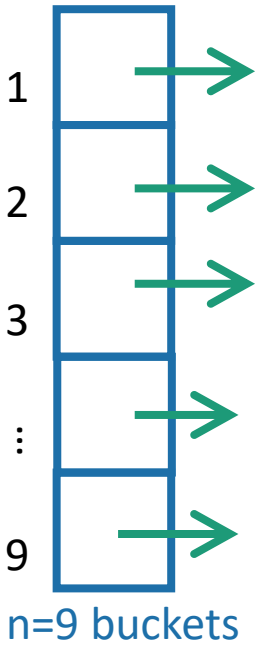
- The previous slide is about hash tables with chaining.
- There's also something called "open addressing"
- Read in CLRS (11.4) if you are interested!




# Aside: Hash tables with open addressing

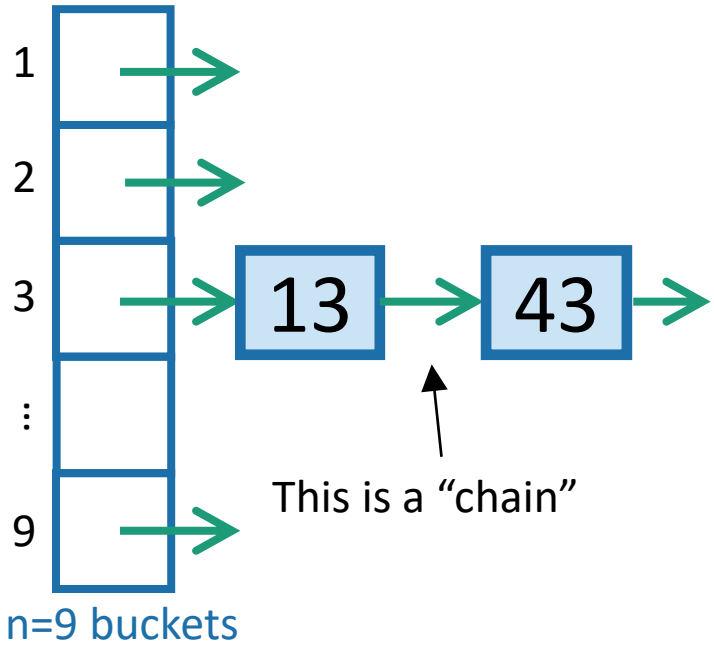
- The previous slide is about hash tables with chaining.
- There's also something called "open addressing"
- Read in CLRS (11.4) if you are interested!

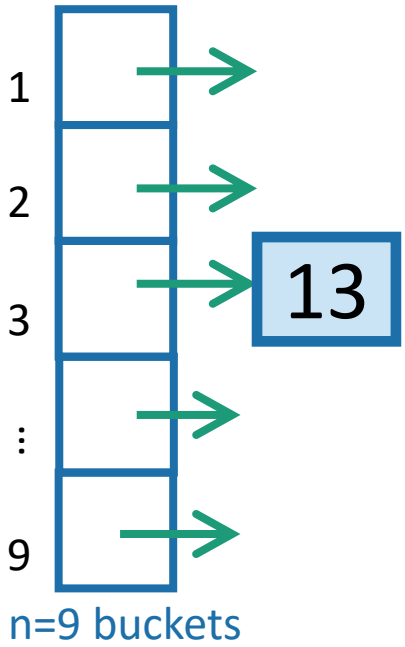



# Aside: Hash tables with open addressing

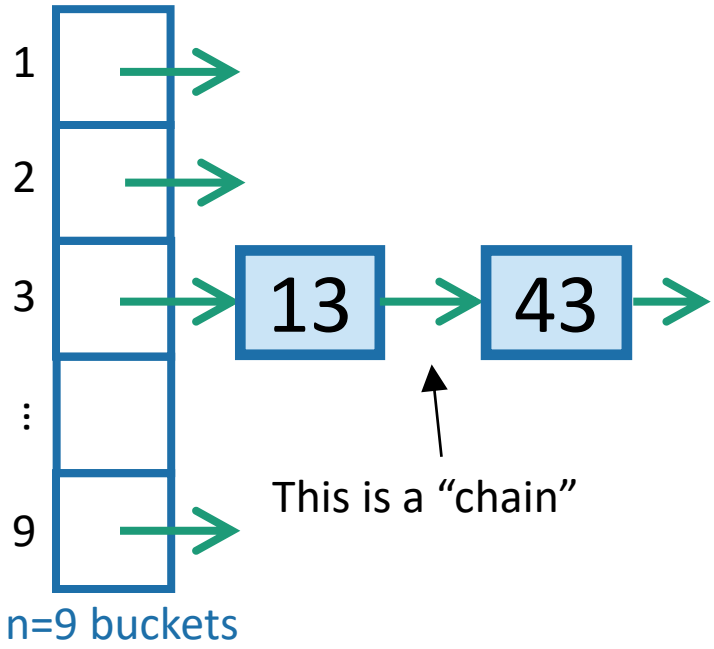
- The previous slide is about hash tables with chaining.
- There's also something called "open addressing"
- Read in CLRS (11.4) if you are interested!

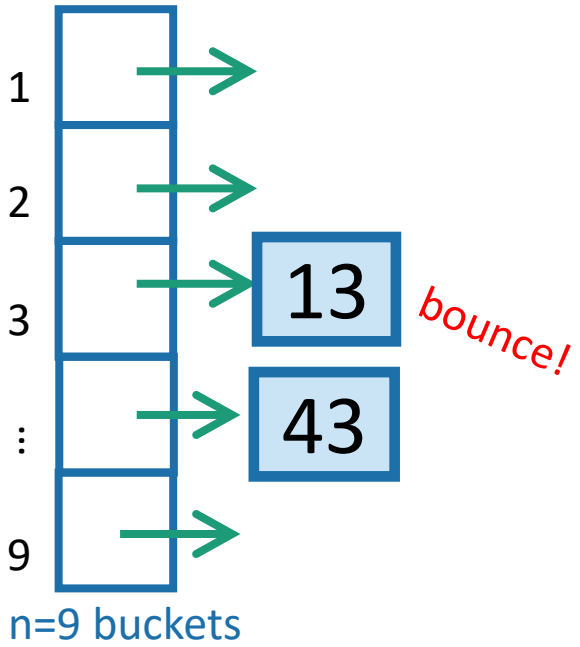



# Aside: Hash tables with open addressing

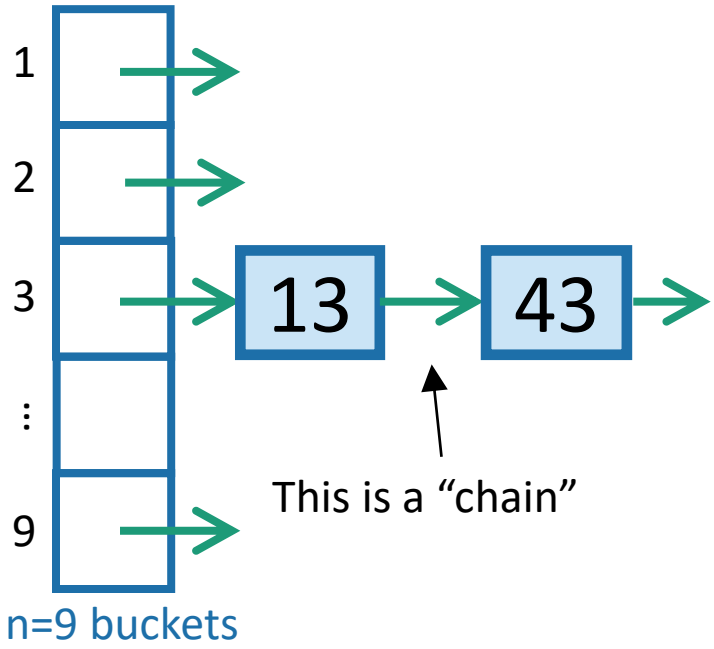
- The previous slide is about hash tables with chaining.
- There's also something called "open addressing"
- Read in CLRS (11.4) if you are interested!

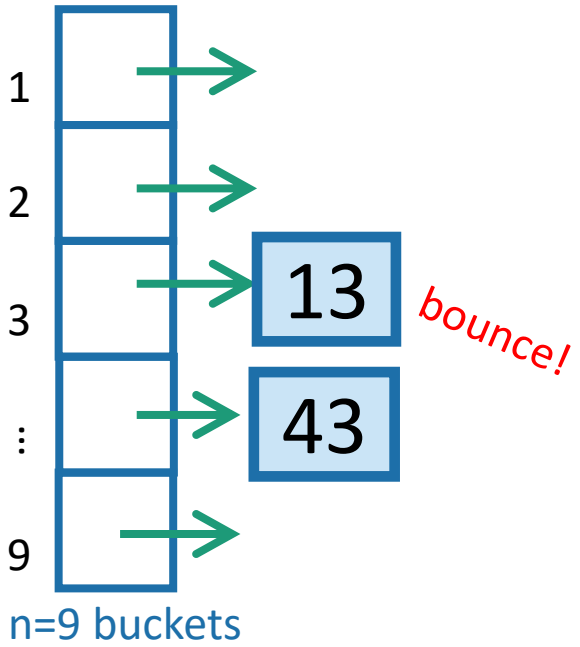



# Aside: Hash tables with open addressing

- The previous slide is about hash tables with chaining.
- There's also something called "open addressing"
- Read in CLRS (11.4) if you are interested!




\end{Aside}

# This is a **hash table** (with chaining)

- Array of n buckets.
- Each bucket stores a linked list.
  - We can insert into a linked list in time O(1)
  - To find something in the linked list takes time O(length(list)).
- $h: U \to \{1, \ldots, n\}$ can be any function:
  - but for concreteness let's stick with h(x) = least significant digit of x.

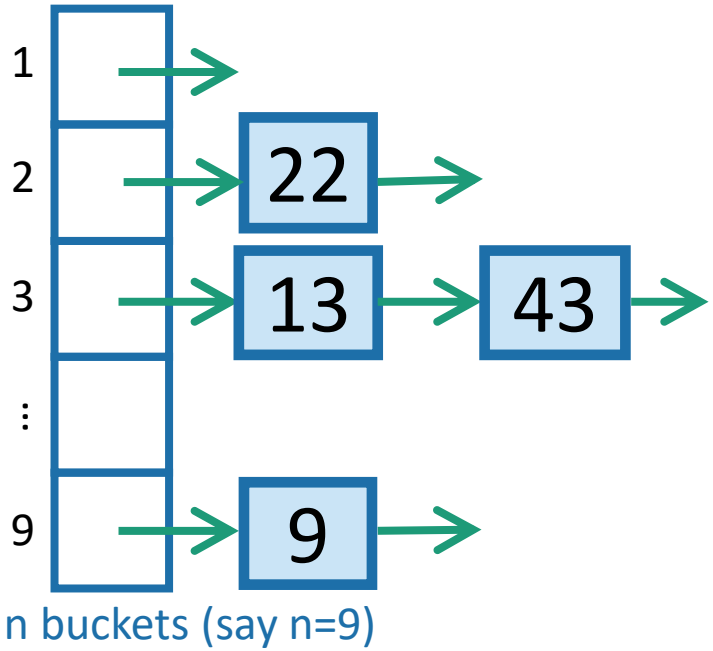
**For demonstration purposes only!**
This is a terrible hash function! Don't use this!

INSERT:

13 22 43 9

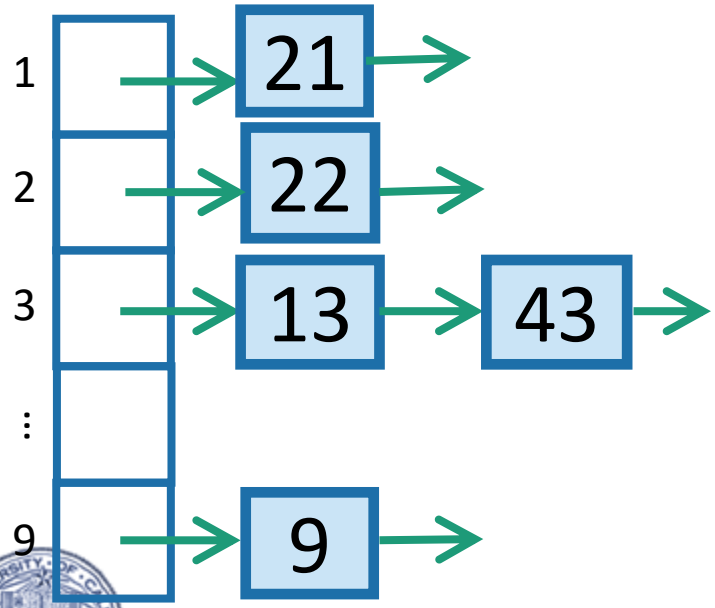
SEARCH 43:

Scan through all the elements in bucket h(43) = 3.



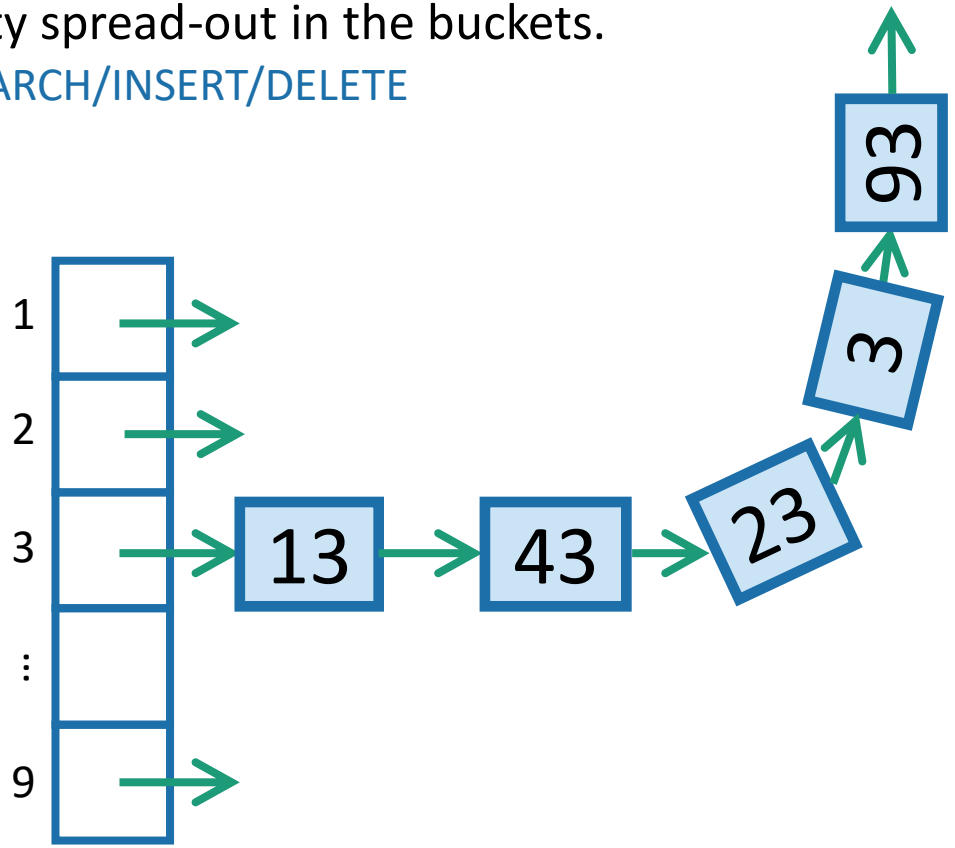
# Sometimes this a good idea
# Sometimes this is a bad idea

- How do we pick that function so that this is a good idea?
  1. We want there to be not many buckets (say, n).
     - This means we don't use too much space
  2. We want the items to be pretty spread-out in the buckets.
     - This means it will be fast to SEARCH/INSERT/DELETE




n=9 buckets

vs.

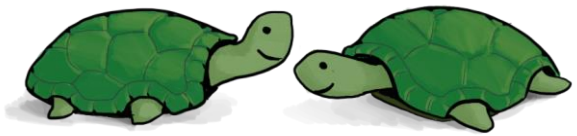



n=9 buckets

# Worst-case analysis

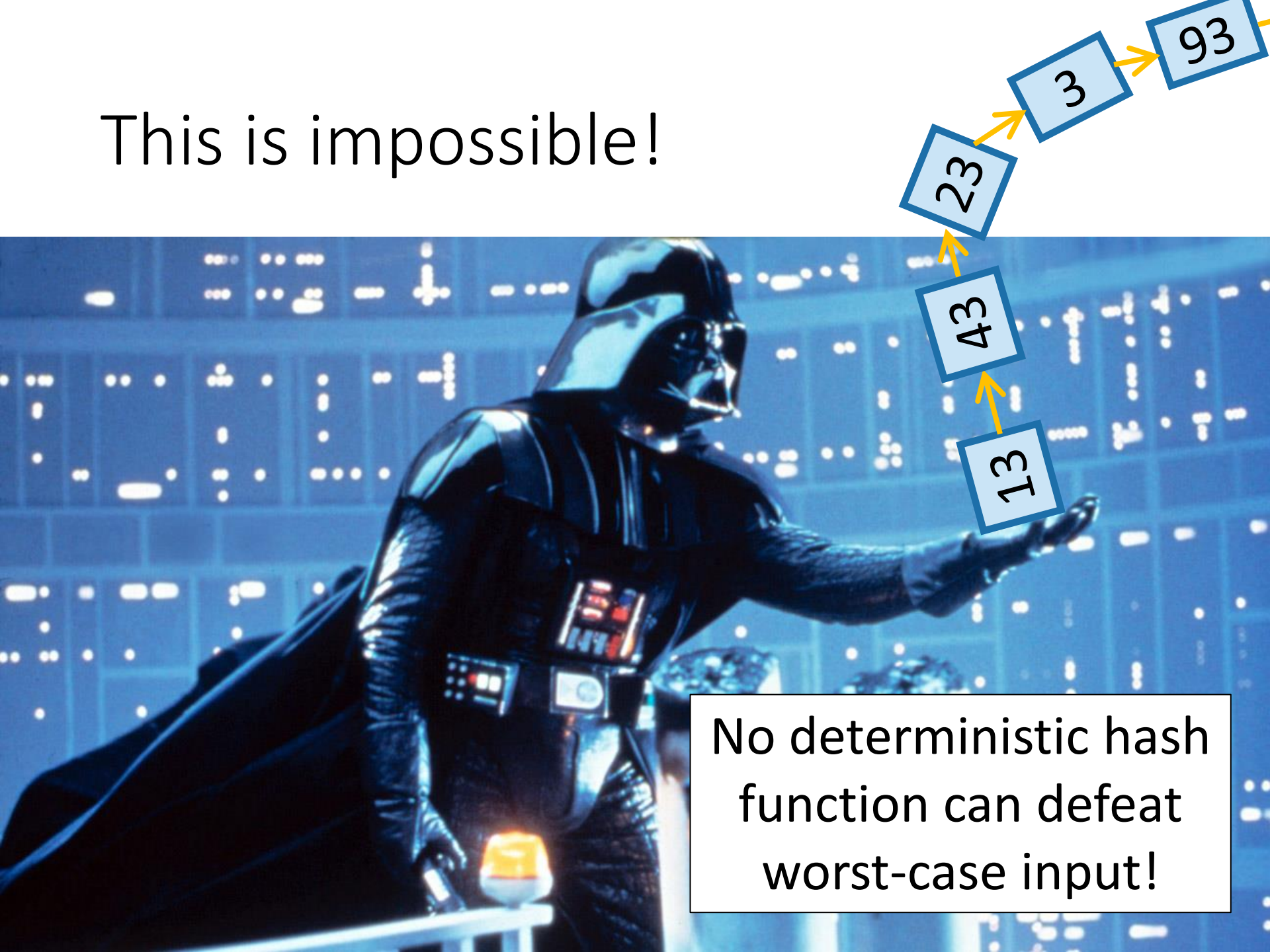
- Goal: Design a function $h: U \to \{1, \dots, n\}$ so that:
  - No matter what input (fewer than n items of U) a bad guy chooses, the buckets will be balanced.
  - Here, balanced means O(1) entries per bucket.
- If we had this, then we'd achieve our dream of O(1) INSERT/DELETE/SEARCH

Can you come up with such a function?



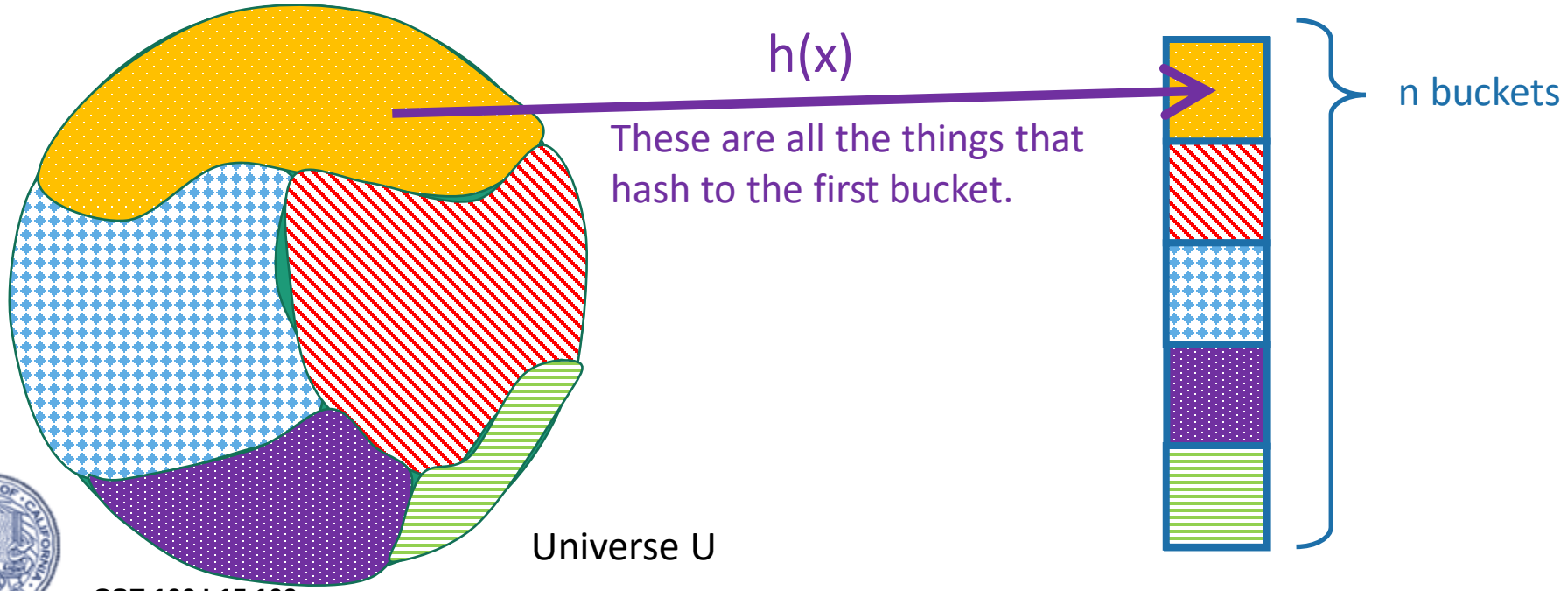
Think-Pair-Share Terrapins

# This is impossible!



# We really can't beat the bad guy here.

- The universe U has M items
- They get hashed into n buckets
- At least one bucket has at least M/n items hashed to it.
- M is waayyyy bigger than n, so M/n is bigger than n.
- **Bad guy chooses n of the items that landed in this very full bucket.**



# Solution:
# Randomness

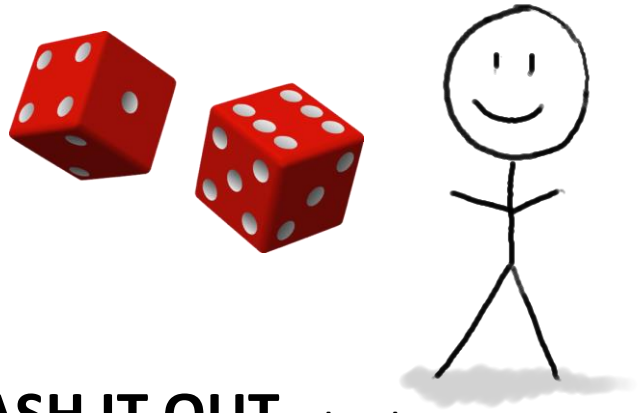
# The game

What does **random** mean here? Uniformly random?

Plucky the pedantic penguin

1. An adversary chooses any n items $u_1, u_2, \ldots, u_n \in U$, and any sequence of INSERT/DELETE/SEARCH operations on those items.

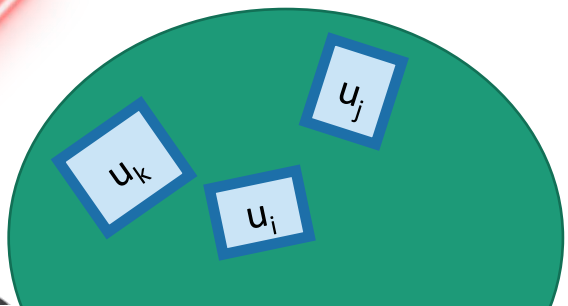
2. You, the algorithm, chooses a **random** hash function $h: U \to \{1, \ldots, n\}$.

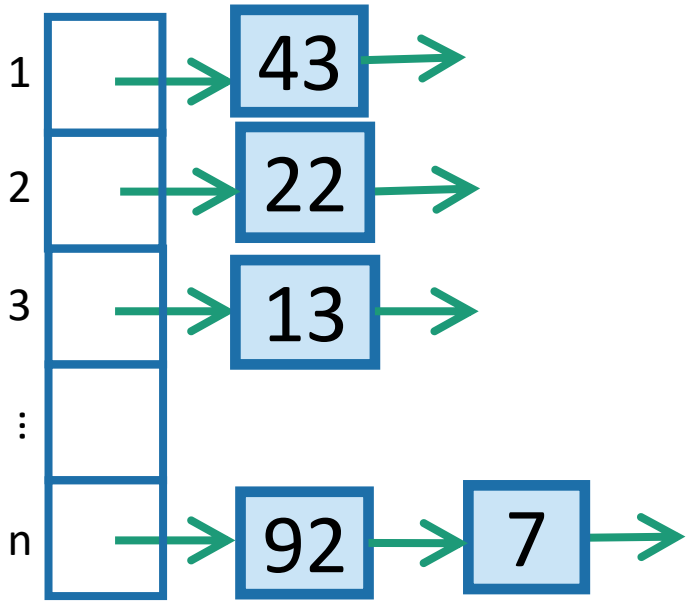


3. **HASH IT OUT** #hashpuns

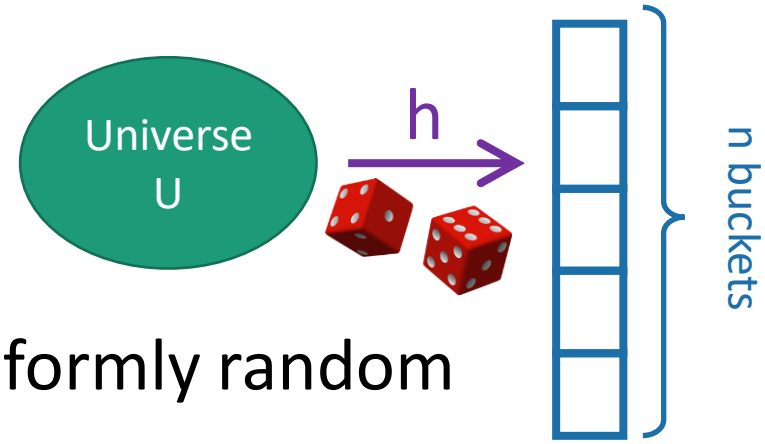
| 13 | 22 | 43 | 92 | 7 |
|---|---|---|---|---|

INSERT 13, INSERT 22, INSERT 43,
INSERT 92, INSERT 7, SEARCH 43,
DELETE 92, SEARCH 7, INSERT 92



# Example




- Say that $h: U \to \{1, \ldots, n\}$ is a uniformly random function.
  - That means that **h(1)** is a **uniformly random** number between 1 and n.
  - **h(2)** is also a **uniformly random** number between 1 and n, independent of h(1).
  - **h(3)** is also a **uniformly random** number between 1 and n, independent of h(1), h(2).
  - …
  - **h(M)** is also a **uniformly random** number between 1 and n, independent of h(1), h(2), …, h(M-1).

# Randomness helps

Intuitively: The bad guy can't foil a hash function that they don't yet know.

Lucky the Lackadaisical Lemur

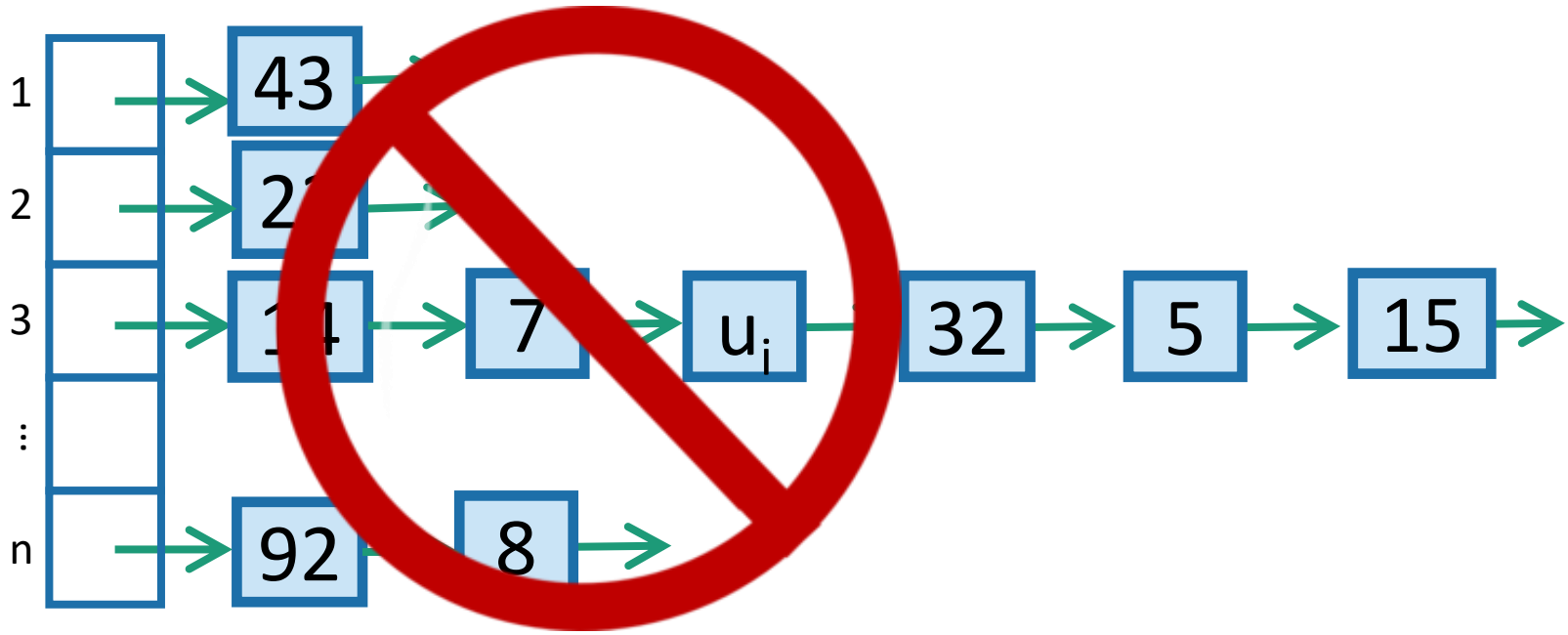
Plucky the Pedantic Penguin

Why not? What if there's some strategy that foils a random function with high probability?

We'll need to do some analysis…

# What do we want?

It's **bad** if lots of items land in $u_i$'s bucket.
So, we **don't want that**.



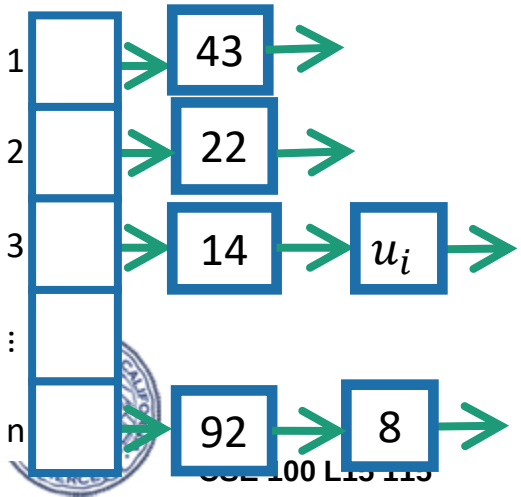

# More precisely

We could replace "2" here with any constant; it would still be good. But "2" will be convenient.

- We want:
  - For all ways a bad guy could choose $u_1, u_2, \ldots, u_n$, to put into the hash table, and for all $i \in \{1, \ldots, n\}$,

    E[ number of items in $u_i$ 's bucket ] $\leq 2$.
- If that were the case:
  - For each INSERT/DELETE/SEARCH operation involving $u_i$,

    E[ time of operation ] = O(1)




This is what we wanted at the beginning of lecture!

# So we want:

- For all i=1, …, n,

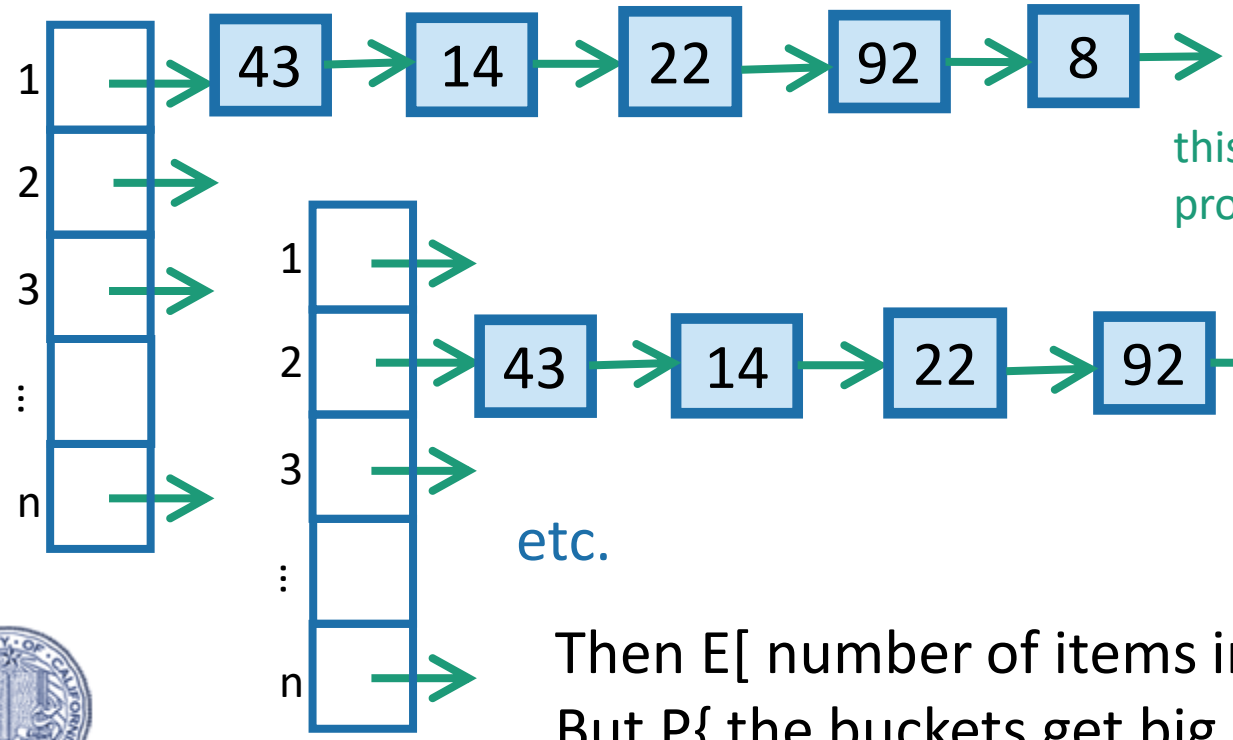
  E[ number of items in $u_i$'s bucket ] $\leq 2$.

# Aside: why not:

- For all i=1,…,n:

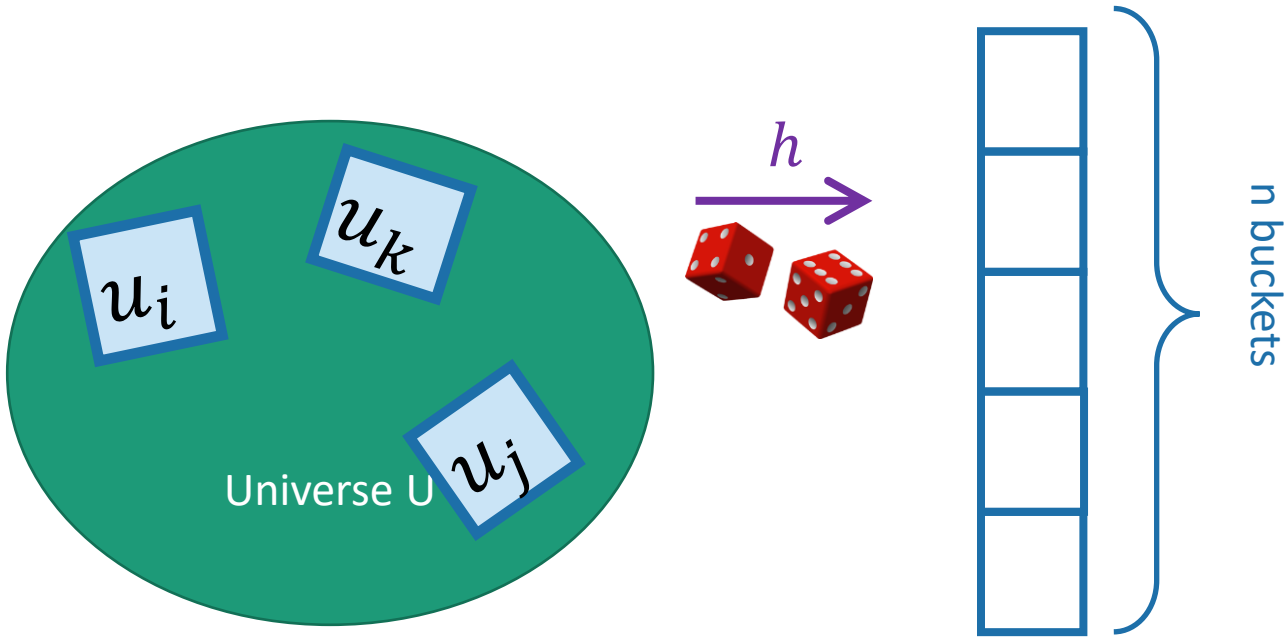
E[ number of items in bucket i ] ≤ ___?

Suppose that:



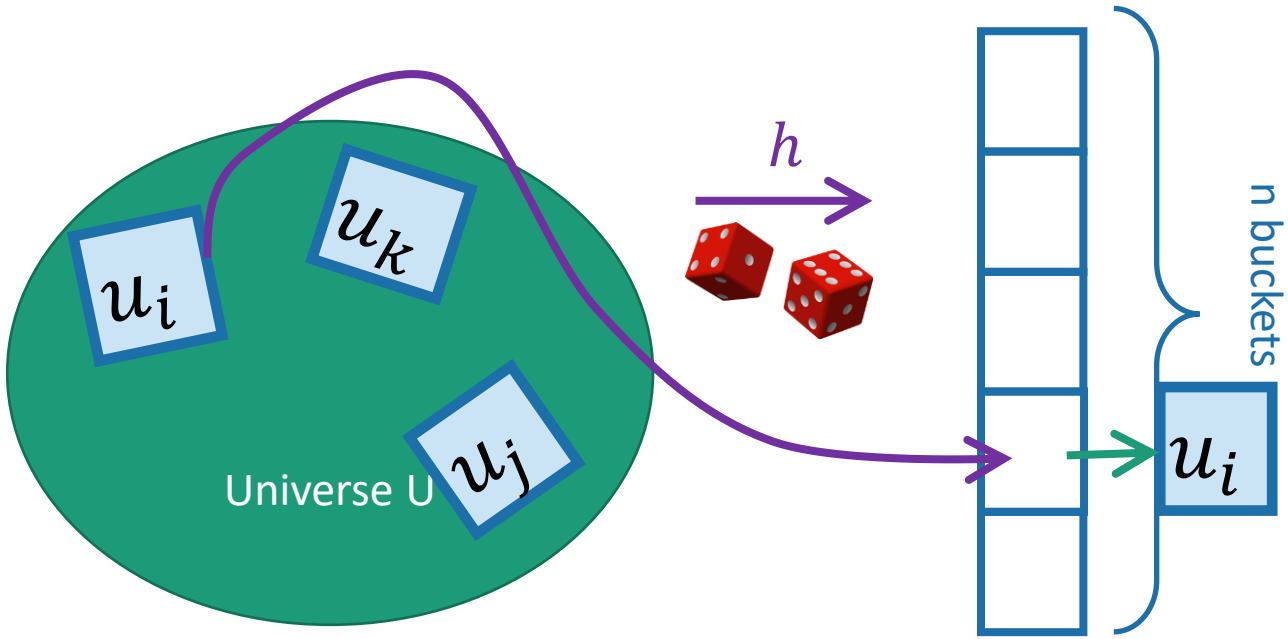

Then E[ number of items in bucket i ] = 1 for all i.
But P{ the buckets get big } = 1.

# Expected number of items in $u_i$'s bucket?



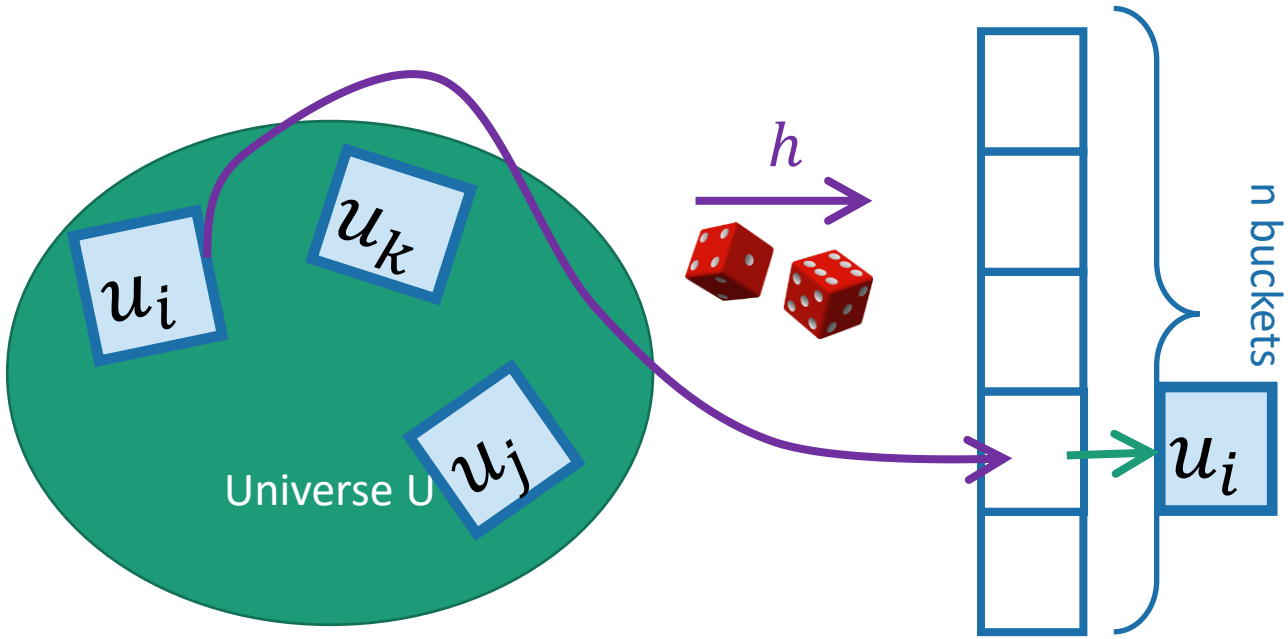

h is uniformly random

# Expected number of items in $u_i$'s bucket?




# Expected number of items in $u_i$'s bucket?

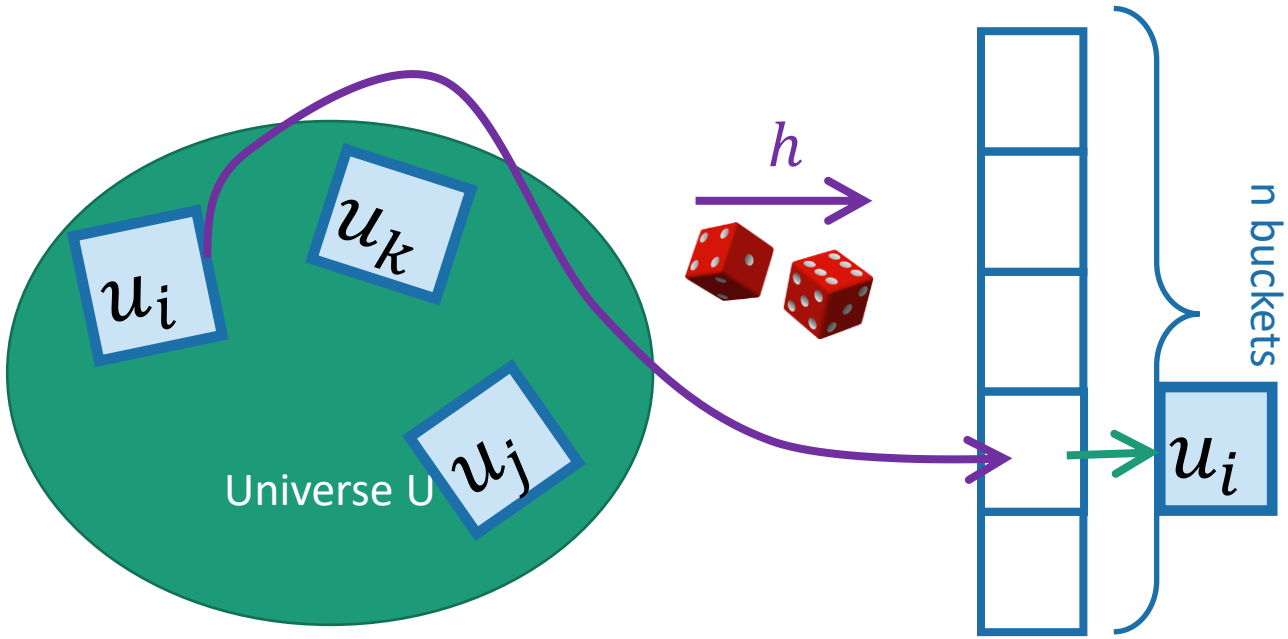
- $E[\ ] = \sum_{j=1}^{n} P\{ h(u_i) = h(u_j)\}$



h is uniformly random

# Expected number of items in $u_i$'s bucket?

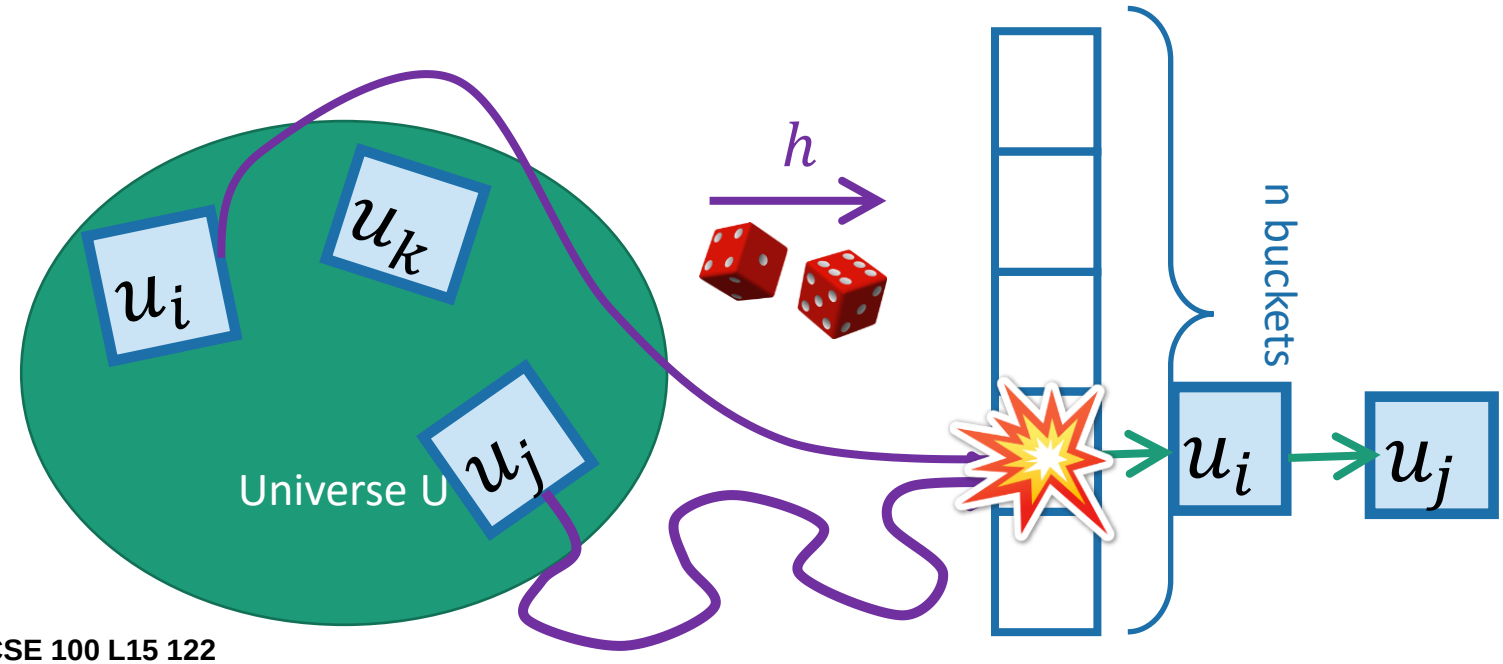
- $E[\ ] = \sum_{j=1}^{n} P\{ h(u_i) = h(u_j)\}$
- $= 1 + \sum_{j \neq i} P\{ h(u_i) = h(u_j)\}$



h is uniformly random

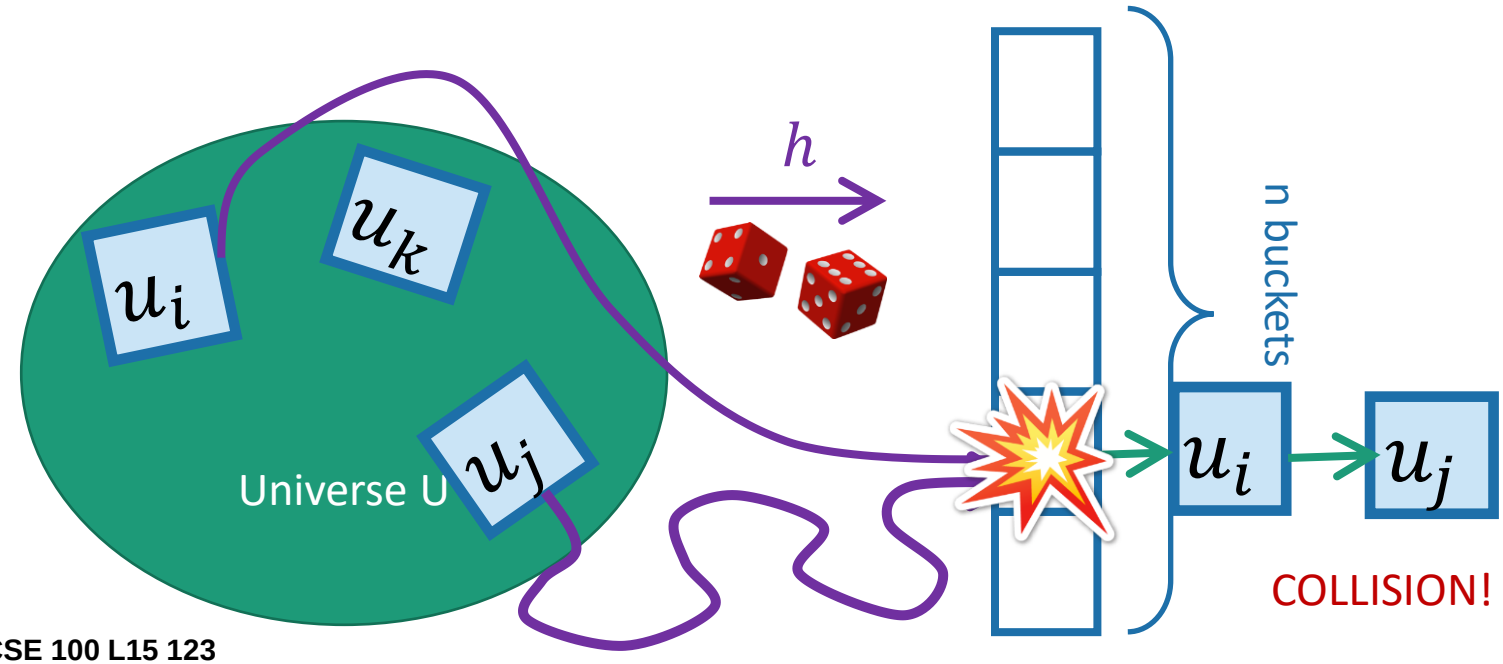
# Expected number of items in $u_i$'s bucket?

- $E[\ ] = \sum_{j=1}^{n} P\{ h(u_i) = h(u_j)\}$
- $= 1 + \sum_{j \neq i} P\{ h(u_i) = h(u_j)\}$



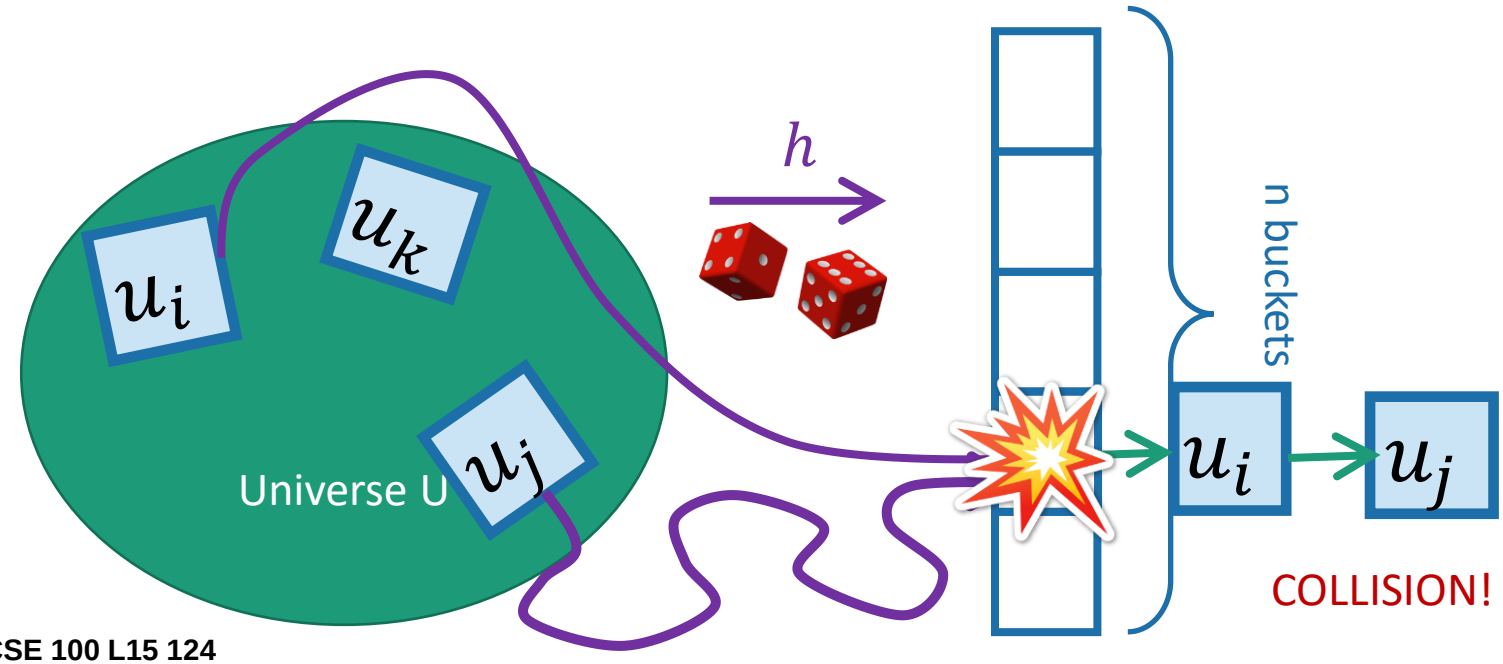
# Expected number of items in $u_i$'s bucket?

- $E[\ ] = \sum_{j=1}^{n} P\{ h(u_i) = h(u_j)\}$
- $= 1 + \sum_{j \neq i} P\{ h(u_i) = h(u_j)\}$



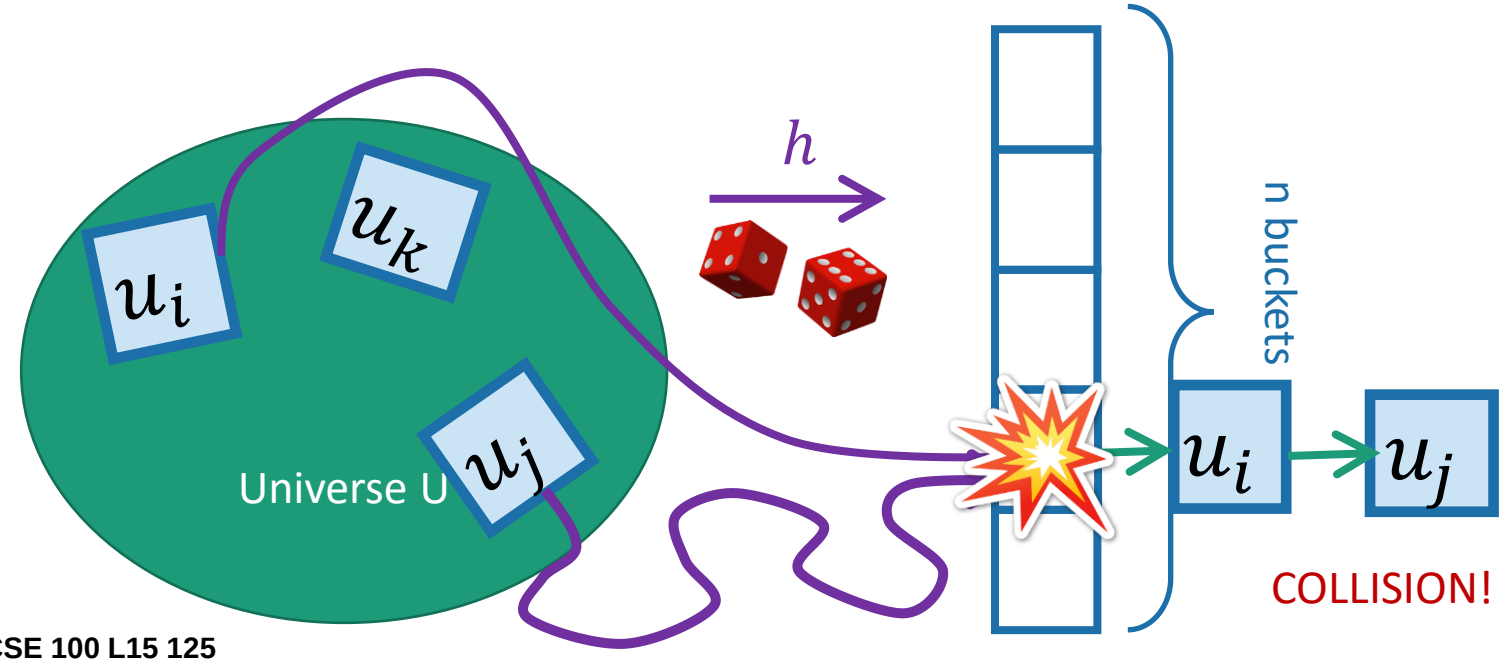
# Expected number of items in $u_i$'s bucket?

- $E[\ ] = \sum_{j=1}^{n} P\{ h(u_i) = h(u_j)\}$
- $= 1 + \sum_{j \neq i} P\{ h(u_i) = h(u_j)\}$
- $= 1 + \sum_{j \neq i} 1/n$



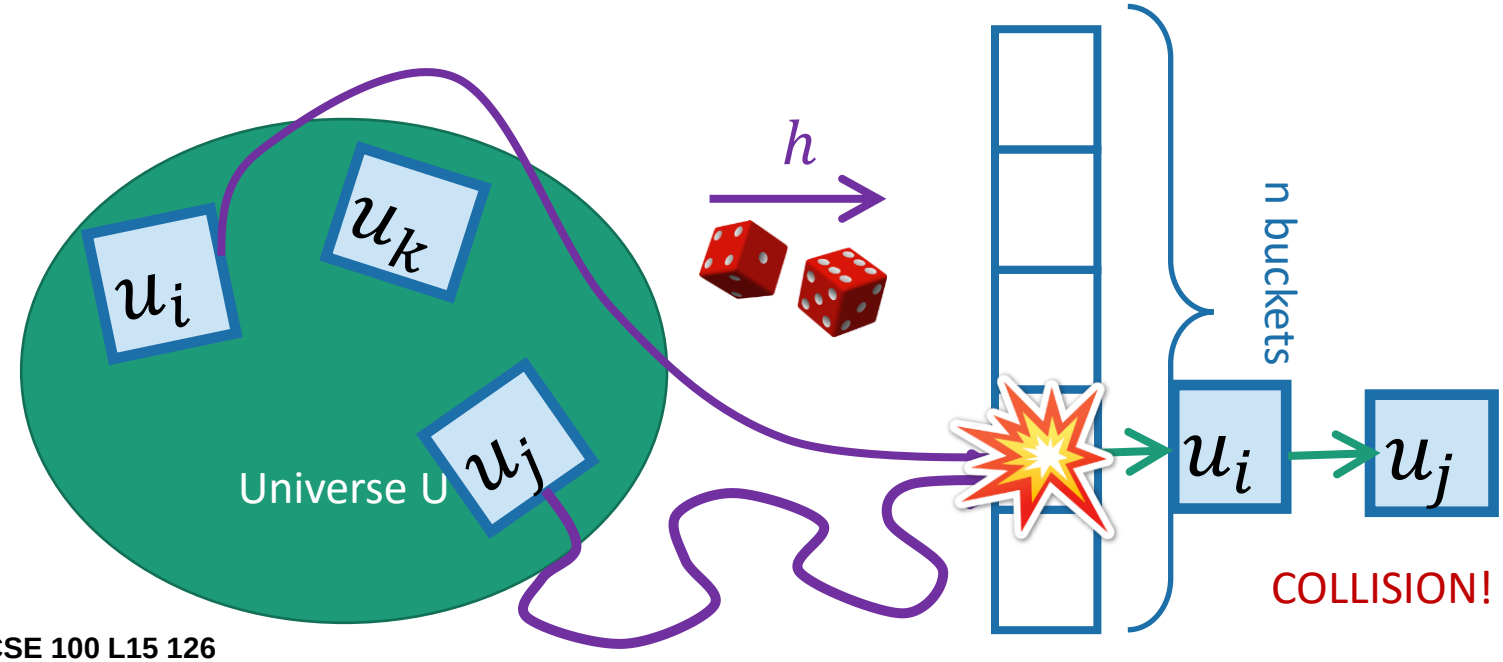
# Expected number of items in $u_i$'s bucket?

- $E[\ ] = \sum_{j=1}^{n} P\{ h(u_i) = h(u_j)\}$
- $= 1 + \sum_{j \neq i} P\{ h(u_i) = h(u_j)\}$
- $= 1 + \sum_{j \neq i} 1/n$ (You need to verify this on your own)



# Expected number of items in $u_i$'s bucket?

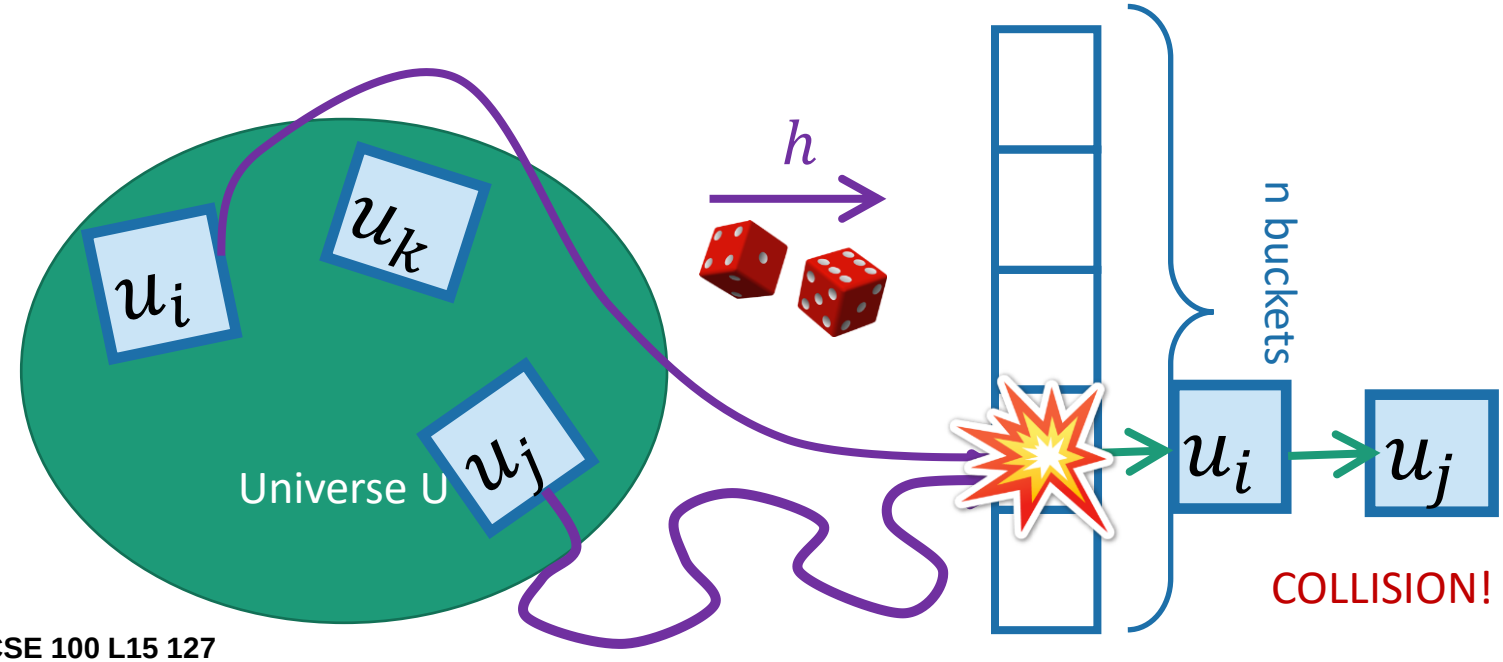
- $E[\ ] = \sum_{j=1}^{n} P\{ h(u_i) = h(u_j)\}$
- $= 1 + \sum_{j \neq i} P\{ h(u_i) = h(u_j)\}$
- $= 1 + \sum_{j \neq i} 1/n$ (You need to verify this on your own)
- $= 1 + \frac{n-1}{n} \leq 2.$



h is uniformly random

# Expected number of items in $u_i$'s bucket?

- $E[\ ] = \sum_{j=1}^{n} P\{ h(u_i) = h(u_j)\}$
- $= 1 + \sum_{j \neq i} P\{ h(u_i) = h(u_j)\}$
- $= 1 + \sum_{j \neq i} 1/n$ (You need to verify this on your own)
- $= 1 + \frac{n-1}{n} \leq 2.$ That's what we wanted!



# That's great!

- We just showed:
  - For all ways a bad guy could choose $u_1, u_2, \ldots, u_n$, to put into the hash table, and for all $i \in \{1, \ldots, n\}$,

    E[ number of items in $u_i$ 's bucket ] $\leq 2$.
- Which implies:
  - No matter what sequence of operations and items the bad guy chooses,

    E[ time of INSERT/DELETE/SEARCH ] = O(1)
- So, our solution is:

  Pick a uniformly random hash function?