CSE 250

Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu
208 Capen Hall

Tree Wrap-Up and Hash Functions

Announcements and Feedback

- PA3 Implementation due Sunday @ 11:59PM
- WA3 coming soon…will include PA3 wrap-up and tree questions

BST Operations

Operation	**BST**	**AVL**	**Red-Black**
`find`	$O(d) = O(n)$	$O(d) = O(\log n)$	$O(d) = O(\log n)$
`insert`	$O(d) = O(n)$	$O(d) = O(\log n)$	$O(d) = O(\log n)$
`remove`	$O(d) = O(n)$	$O(d) = O(\log n)$	$O(d) = O(\log n)$

The tree operations on a BST are always **$O(d)$** (they involve a constant number of trips from root to leaf at most).

The balanced varieties (AVL and Red-Black) constrain the depth

Constraining Tree Depth

AVL Trees

Keep tree **balanced**: subtrees **+/-1** of each other in height

- Add a field to track amount of "imbalance"
- If imbalance exceeds +/-1 perform rotations to fix

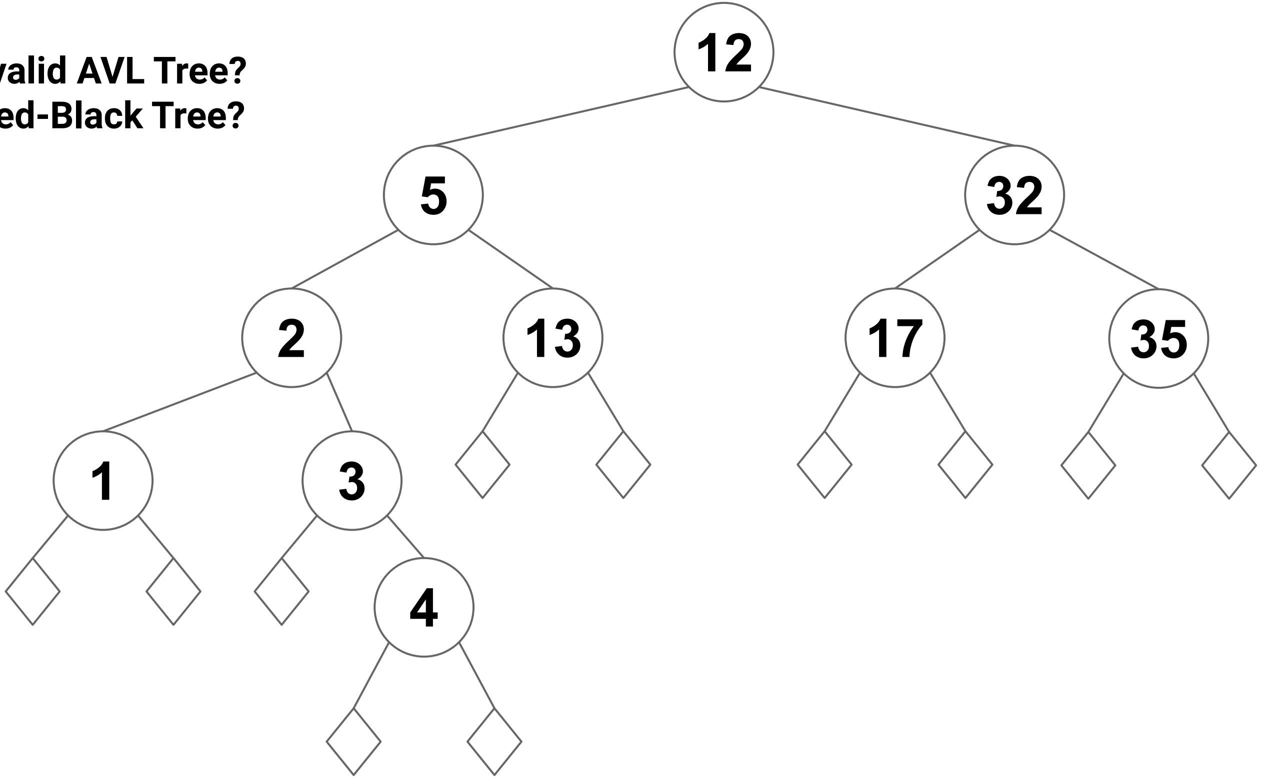
Red-Black Trees

Keep leaves at some minimum depth (***d*/2**)

- Add a color to each node marking it as "red" or "black"
    a. Keep # of black nodes = on every path from leaf to root
    b. Don't let red nodes have red parents
- If either rule is broken, rotate and recolor to fix

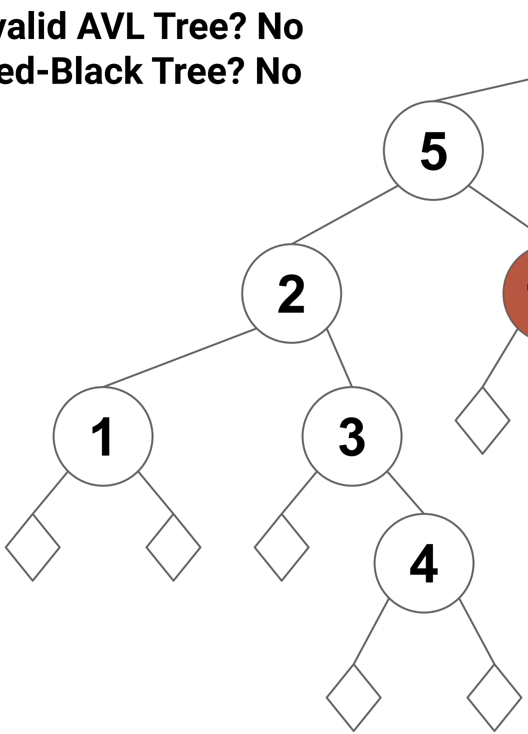
Example

Is this a valid AVL Tree?
A valid Red-Black Tree?



Example

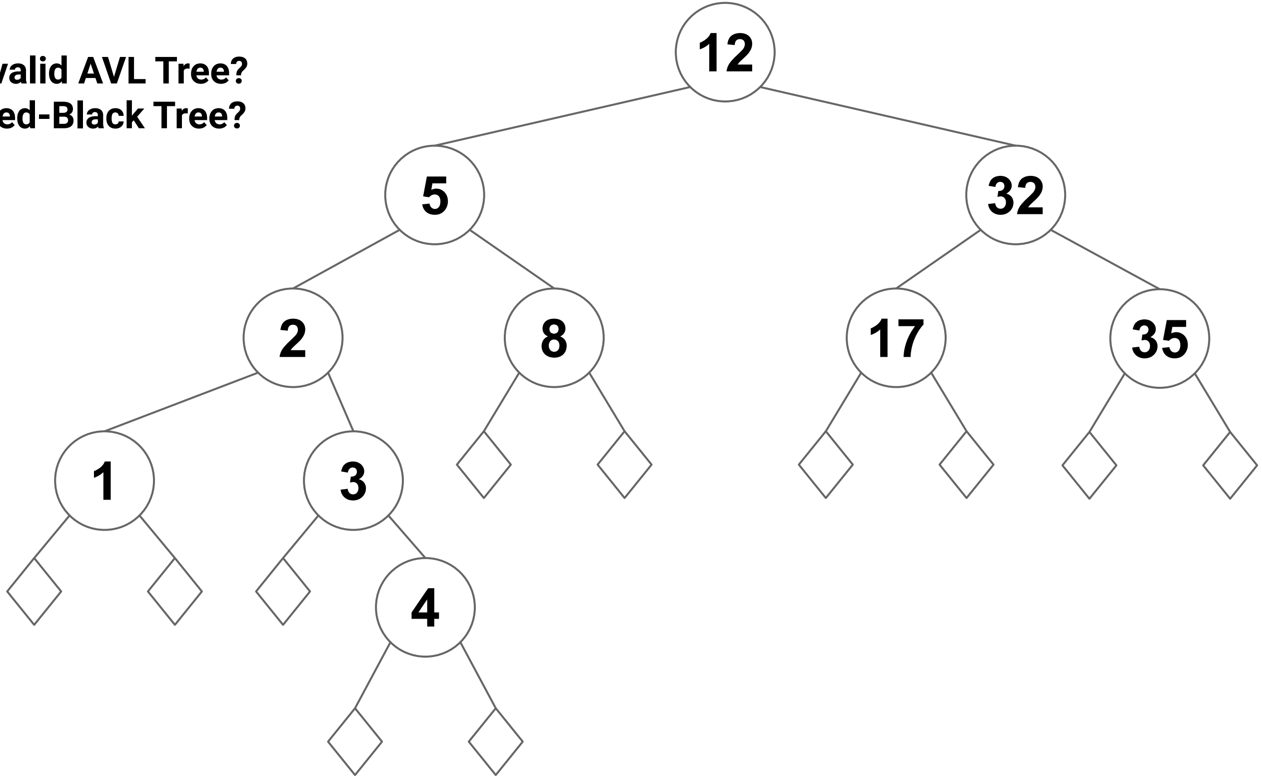
Is this a valid AVL Tree? No
A valid Red-Black Tree? No



The 13 is larger than 12, but left of 12. Don't forget, AVL and Red-Black trees are both BSTs!

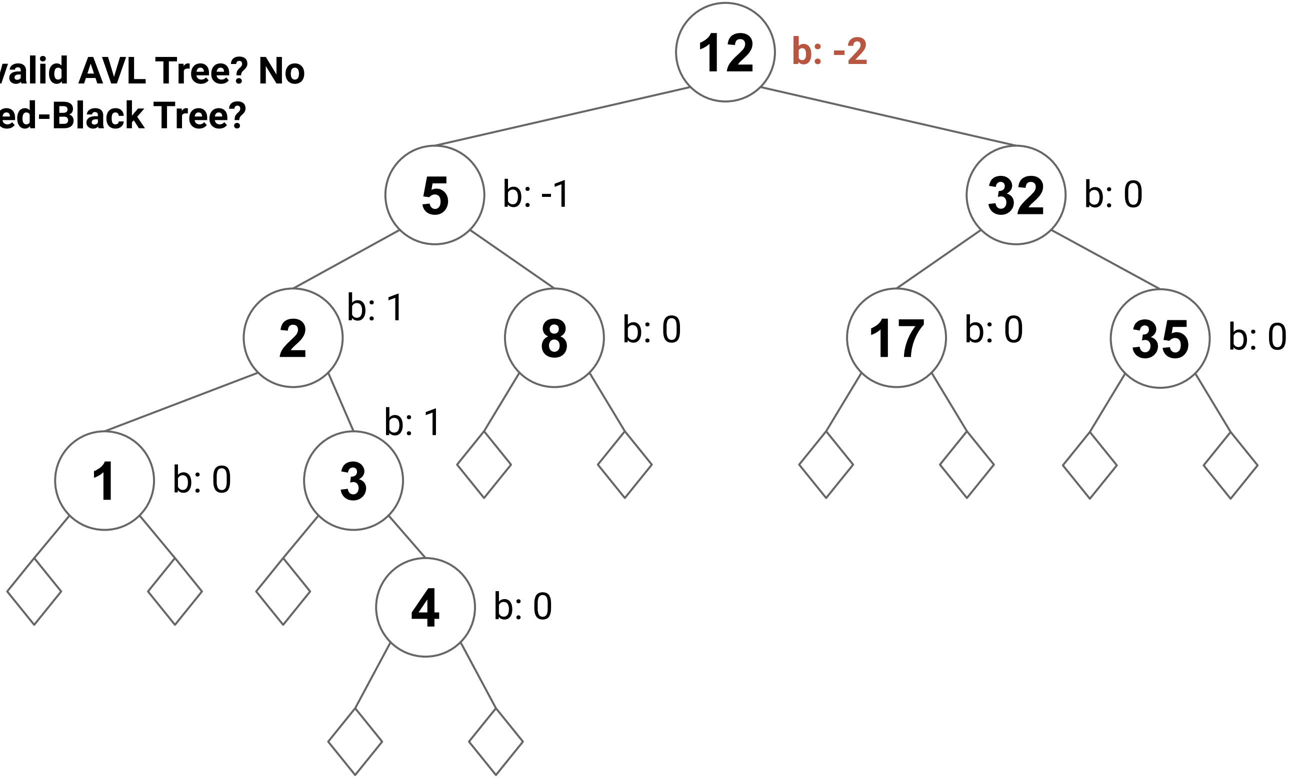
Example

Is this a valid AVL Tree?
A valid Red-Black Tree?



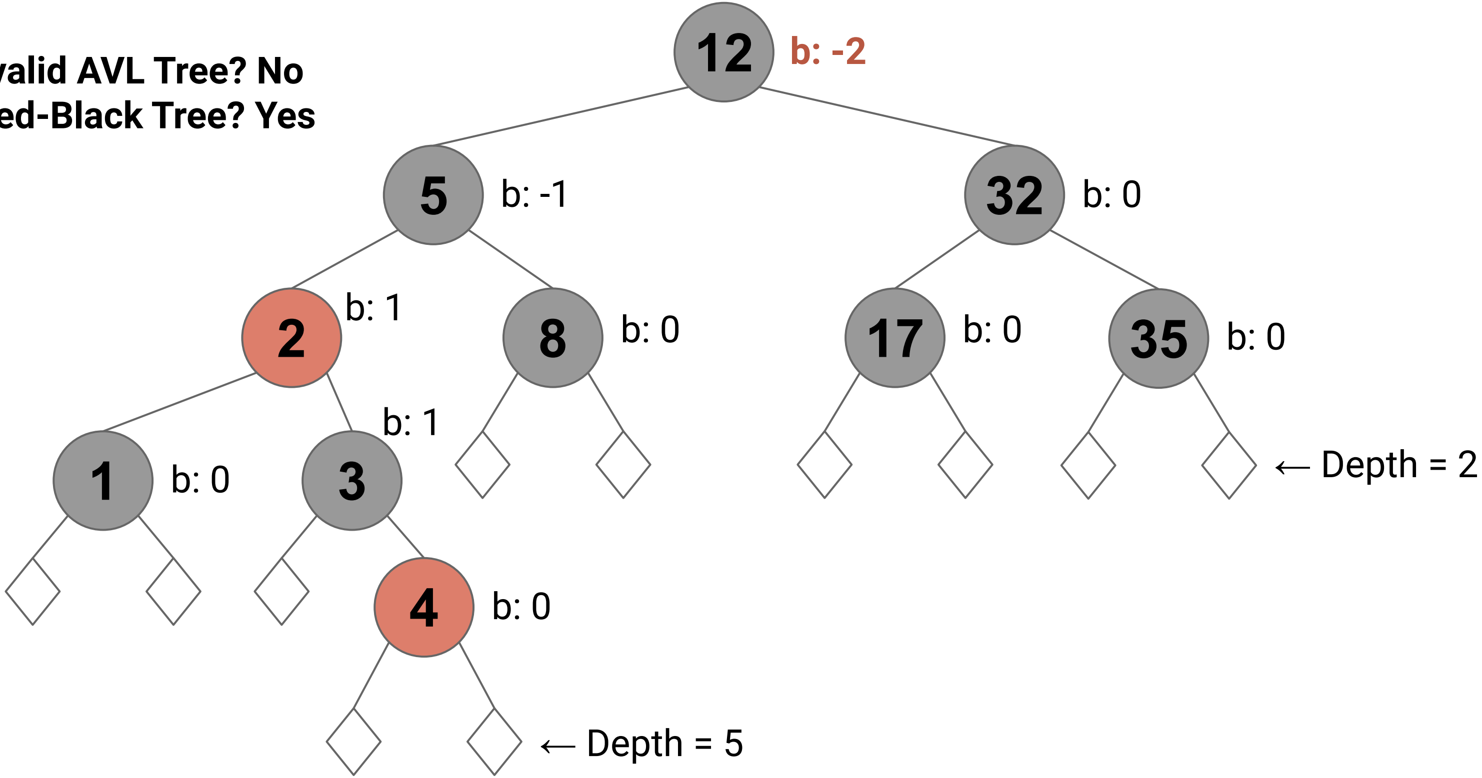
Example

Is this a valid AVL Tree? No
A valid Red-Black Tree?



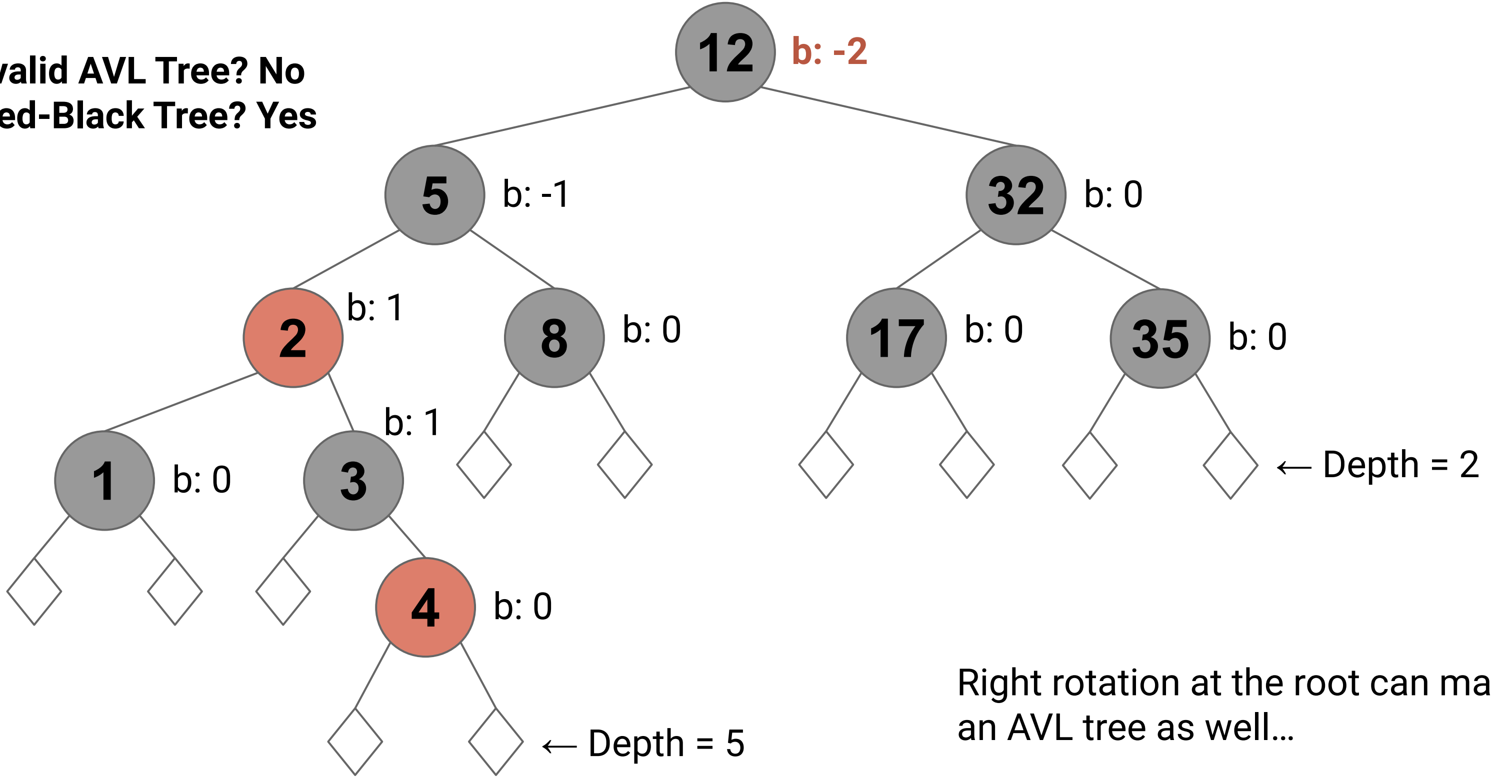
Example

Is this a valid AVL Tree? No
A valid Red-Black Tree? Yes



Example

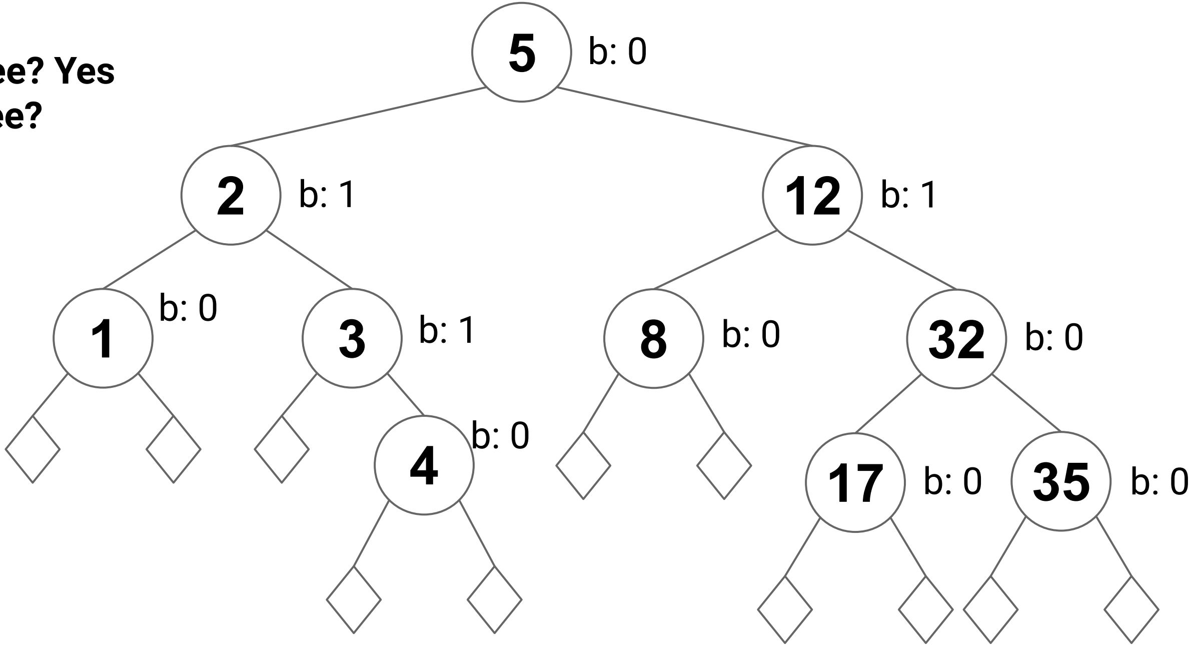
Is this a valid AVL Tree? No
A valid Red-Black Tree? Yes



Right rotation at the root can make this an AVL tree as well…

Example

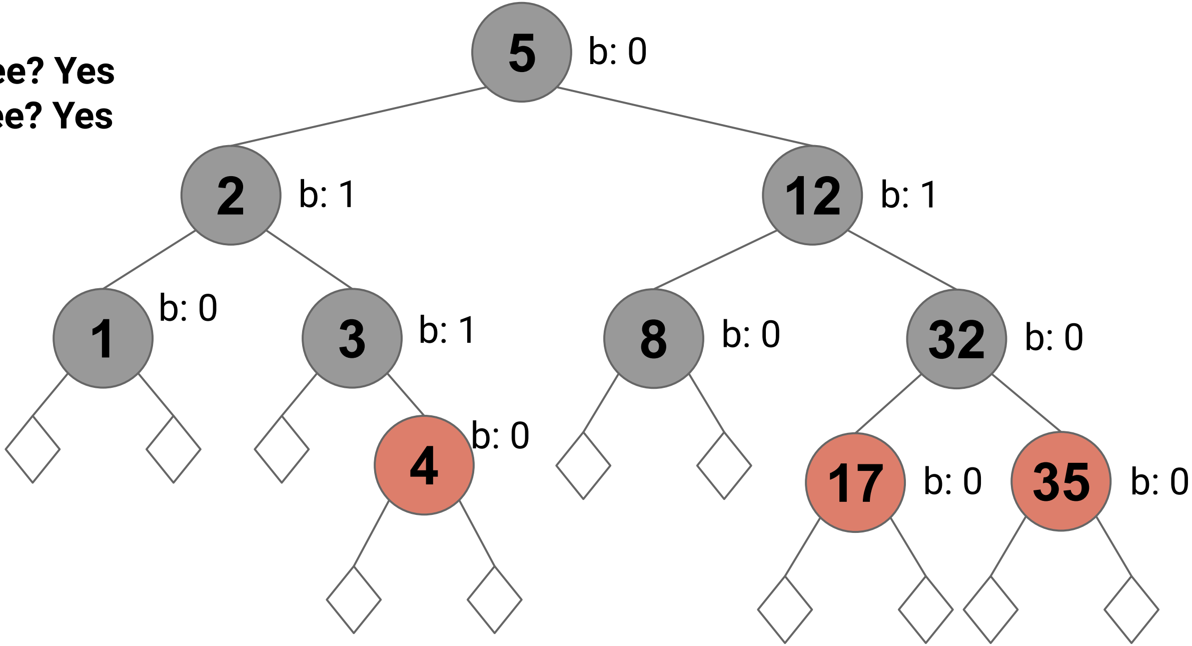
Is this a valid AVL Tree? Yes
A valid Red-Black Tree?



Right rotation at the root can make this an AVL tree as well…

Example

Is this a valid AVL Tree? Yes
A valid Red-Black Tree? Yes



It's also still a Red-Black tree… EVERY AVL tree can be colored with a valid Red-Black coloring.
(But not every Red-Black tree meets AVL constraints)

Now how can we use trees…

The `mutable.Set[T]` ADT

`add(element: T): Unit`

Store one copy of **`element`** if not already present

`apply(element: T): Boolean`

Return true if **`element`** is present in the set

`remove(element: T): Boolean`

Remove **`element`** if present, or return false if not

Implementing Sets

We've seen a few data structures we could use to implement the Set ADT:

- Linked Lists
- ArrayBuffers
- BSTs

What do these implementations look like and how do they perform?

Implementing `Set.add`

Implementing `add`:

- With a LinkedList?

Implementing `Set.add`

Implementing `add`:

- With a LinkedList? Just prepend the element in ***O*(1)** time (could also store it in sorted order in ***O*(*n*)** time)

Implementing `Set.add`

Implementing `add`:

- With a LinkedList? Just prepend the element in $\boldsymbol{O(1)}$ time (could also store it in sorted order in $\boldsymbol{O(n)}$ time)
- With an ArrayBuffer?

Implementing `Set.add`

Implementing `add`:

- With a LinkedList? Just prepend the element in ***O*(1)** time (could also store it in sorted order in ***O*(*n*)** time)
- With an ArrayBuffer? Just append the element in amortized ***O*(1)** time (could also store in sorted order in ***O*(*n*)** time)

Implementing `Set.add`

Implementing `add`:

- With a LinkedList? Just prepend the element in ***O*(1)** time (could also store it in sorted order in ***O*(*n*)** time)
- With an ArrayBuffer? Just append the element in amortized ***O*(1)** time (could also store in sorted order in ***O*(*n*)** time)
- With a BST?

Implementing `Set.add`

Implementing `add`:

- With a LinkedList? Just prepend the element in $O(1)$ time (could also store it in sorted order in $O(n)$ time)
- With an ArrayBuffer? Just append the element in amortized $O(1)$ time (could also store in sorted order in $O(n)$ time)
- With a BST? Add the element to the tree in $O(d)$ time…

Implementing `Set.add`

Implementing `add`:

- With a LinkedList? Just prepend the element in ***O*(1)** time (could also store it in sorted order in ***O*(*n*)** time)
- With an ArrayBuffer? Just append the element in amortized ***O*(1)** time (could also store in sorted order in ***O*(*n*)** time)
- With a **Balanced** BST? Add the element to the tree in ***O*(*d*) = *O*(log *n*)**

Implementing `Set.apply`

Implementing `apply`:

- With a LinkedList?

Implementing `Set.apply`

Implementing `apply`:

- With a LinkedList? Search for the element in ***O*(*n*)** time (still ***O*(*n*)** even if the list is sorted)

Implementing `Set.apply`

Implementing `apply`:

- With a LinkedList? Search for the element in ***O*(*n*)** time (still ***O*(*n*)** even if the list is sorted)
- With an ArrayBuffer?

Implementing `Set.apply`

Implementing `apply`:

- With a LinkedList? Search for the element in ***O*(*n*)** time (still ***O*(*n*)** even if the list is sorted)
- With an ArrayBuffer? Search for the element in ***O*(*n*)** time (can search in ***O*(log *n*)** time if the ArrayBuffer is sorted)

Implementing `Set.apply`

Implementing `apply`:

- With a LinkedList? Search for the element in ***O*(*n*)** time (still ***O*(*n*)** even if the list is sorted)
- With an ArrayBuffer? Search for the element in ***O*(*n*)** time (can search in ***O*(log *n*)** time if the ArrayBuffer is sorted)
- With a **Balanced** BST?

Implementing `Set.apply`

Implementing `apply`:

- With a LinkedList? Search for the element in ***O*(*n*)** time (still ***O*(*n*)** even if the list is sorted)
- With an ArrayBuffer? Search for the element in ***O*(*n*)** time (can search in ***O*(log *n*)** time if the ArrayBuffer is sorted)
- With a **Balanced** BST? Find the element ***O*(*d*) = *O*(log *n*)**

Implementing `Set.remove`

Implementing `remove`:

- With a LinkedList?

Implementing `Set.remove`

Implementing `remove`:

- With a LinkedList? Search for the element in $\boldsymbol{O(n)}$ time, remove in $\boldsymbol{O(1)}$

Implementing `Set.remove`

Implementing `remove`:

- With a LinkedList? Search for the element in ***O*(*n*)** time, remove in ***O*(1)**
- With an ArrayBuffer?

Implementing `Set.remove`

Implementing `remove:`

- With a LinkedList? Search for the element in ***O*(*n*)** time, remove in ***O*(1)**
- With an ArrayBuffer? Search for the element in ***O*(*n*)** or ***O*(log *n*)** time but remove in ***O*(*n*)** regardless (have to shift potentially ***n*** elements)

Implementing `Set.remove`

Implementing `remove`:

- With a LinkedList? Search for the element in ***O*(*n*)** time, remove in ***O*(1)**
- With an ArrayBuffer? Search for the element in ***O*(*n*)** or ***O*(log *n*)** time but remove in ***O*(*n*)** regardless (have to shift potentially ***n*** elements)
- With a **Balanced** BST?

Implementing `Set.remove`

Implementing `remove:`

- With a LinkedList? Search for the element in ***O*(*n*)** time, remove in ***O*(1)**
- With an ArrayBuffer? Search for the element in ***O*(*n*)** or ***O*(log *n*)** time but remove in ***O*(*n*)** regardless (have to shift potentially ***n*** elements)
- With a **Balanced** BST? Remove the element ***O*(*d*) = *O*(log *n*)**

Implementing Set

We can implement `Set` (and `Bag`) with a Balanced Binary Tree to give ***O*(log *n*)** runtime for all operations.

What about `Map`?

The `mutable.Set[T]` ADT and Maps

`add(element: T): Unit`

Store one copy of **`element`** if not already present

`apply(element: T): Boolean`

Return true if **`element`** is present in the set

`remove(element: T): Boolean`

Remove **`element`** if present, or return false if not

`Maps` are like Sets, but where **`T`** is a 2-tuple: (key, value)

The identity of the **`element`** is determined by key

The `Map[K,V]` ADT

`add(key: K, value: V): Unit                // AKA put(...)`

Insert **`(key, value)`** into the map. If **`key`** already exists, replace it.

`apply(key: K): V                                // AKA get(...)`

Return the value corresponding to **`key`**

`remove(key: K): V`

Remove the element associated with **`key`** and return the value

`Map` Implementations

`Map[K,V]` as a **Sorted Sequence**

- `apply`
- `add`
- `remove`

`Map[K,V]` as a **balanced Binary Search Tree**

- `apply`
- `add`
- `remove`

`Map` Implementations

`Map[K,V]` as a **Sorted Sequence**

- `apply` **$O(\log(n))$ for Array, $O(n)$ for Linked List**
- `add` **$O(n)$**
- `remove` **$O(n)$**

`Map[K,V]` as a **balanced Binary Search Tree**

- `apply`
- `add`
- `remove`

`Map` Implementations

`Map[K,V]` as a **Sorted Sequence**

- `apply` **$O(\log(n))$ for Array, $O(n)$ for Linked List**
- `add` $O(n)$
- `remove` $O(n)$

`Map[K,V]` as a **balanced Binary Search Tree**

- `apply` $O(\log(n))$
- `add` $O(\log(n))$
- `remove` $O(\log(n))$

`Map` Implementations

`Map[K,V]` as a **Sorted Sequence**

- `apply` **$O(\log(n))$ for Array, $O(n)$ for Linked List**
- `add` $O(n)$
- `remove` $O(n)$

`Map[K,V]` as a **balanced Binary Search Tree**

- `apply` $O(\log(n))$
- `add` $O(\log(n))$
- `remove` $O(\log(n))$

Remember: a `Map` is just a `Set` of tuples, so these runtimes are due to the same implementations we discussed for `Sets` in previous slides

Finding Items

When implementing these operations with a BST where is most of "cost" of each algorithm coming from?

Finding Items

When implementing these operations with a BST where is most of "cost" of each algorithm coming from? **Finding the element**

apply => **find the element**
add => **find the insertion point**, then add (the add is often O(1))
remove => **find the element**, then remove (the remove is often O(1))

Finding Items

When implementing these operations with a BST where is most of "cost" of each algorithm coming from? **Finding the element**

apply => **find the element**
add => **find the insertion point**, then add (the add is often O(1))
remove => **find the element**, then remove (the remove is often O(1))

What if we could just...skip the find step?
What if we knew exactly where the element would be?

Assigning Bins

Which data structure has constant lookup if we know where our element is in a sequence?

Assigning Bins

Which data structure has constant lookup if we know where our element is in a sequence? **An Array**

Assigning Bins

Which data structure has constant lookup if we know where our element is in a sequence? **An Array**

Idea: What if we could assign each record to a location in an Array

- Create and array of size **N**
- Pick an ***O*(1)** function to assign each record a number in **[0,N)**
    - ie: creating a set of movies stored by first letter of title, Movie →[0,26)

Assigning Bins

A	B	…	F	G	H	…	Z

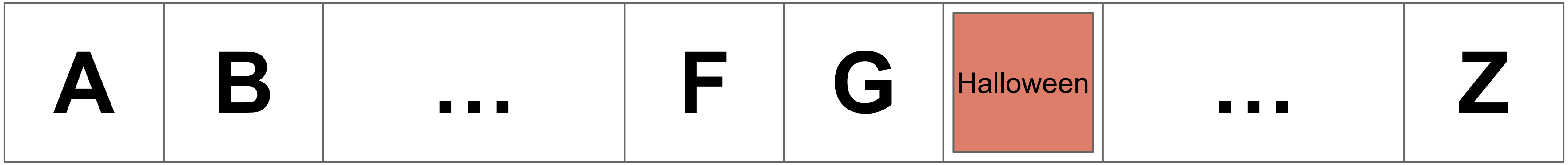
Assigning Bins

`add("Halloween")`

A	B	…	F	G	H	…	Z

Assigning Bins

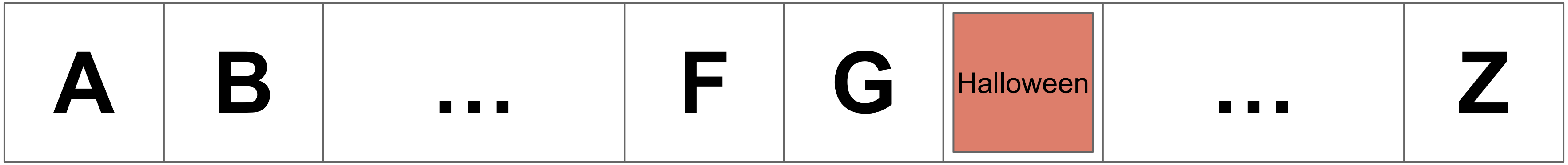
`add("Halloween")` → "Halloween"[0] == "H" == 7



Assigning Bins

`add("Halloween")` → "Halloween"[0] == "H" == 7

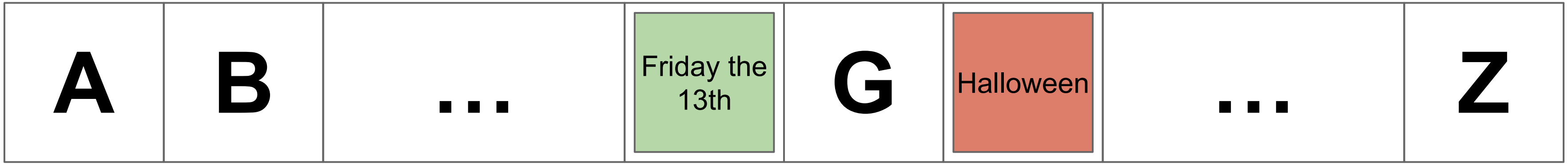
This computation is ***O*(1)**



A	B	…	F	G	Halloween	…	Z

Assigning Bins

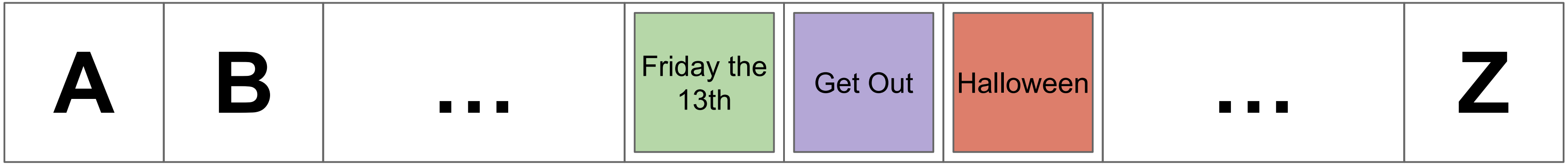
`add("Friday the 13th")` → "Friday the 13th"[0] == "F" == 5



A	B	…	Friday the 13th	G	Halloween	…	Z

Assigning Bins

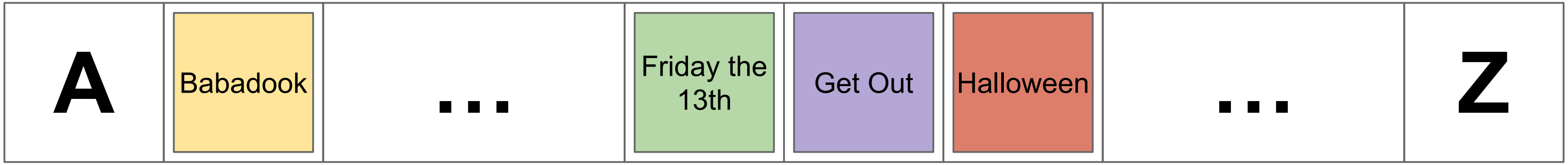
`add("Get Out")` → "Get Out"[0] == "G" == 6



A	**B**	**…**	Friday the 13th	Get Out	Halloween	**…**	**Z**

Assigning Bins

`add("Babadook")` → "Babadook"[0] == "B" == 1

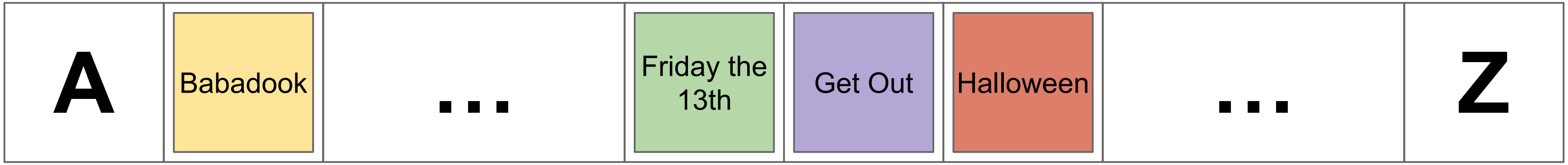


A	Babadook	**...**	Friday the 13th	Get Out	Halloween	**...**	**Z**

Assigning Bins

`find("Get Out")` → "Get Out"[0] == "G" == 6

Find in constant time!

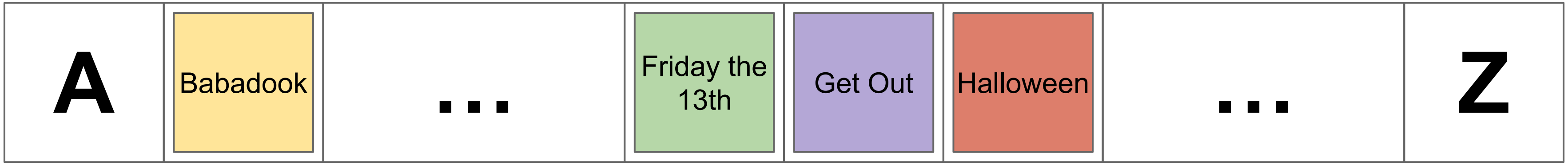


A	Babadook	…	Friday the 13th	Get Out	Halloween	…	**Z**

Assigning Bins

`find("Scream")` → "Scream"[0] == "S" == 18

Determine that "Scream" is not in the Set in constant time!



A

Babadook

...

Friday the 13th

Get Out

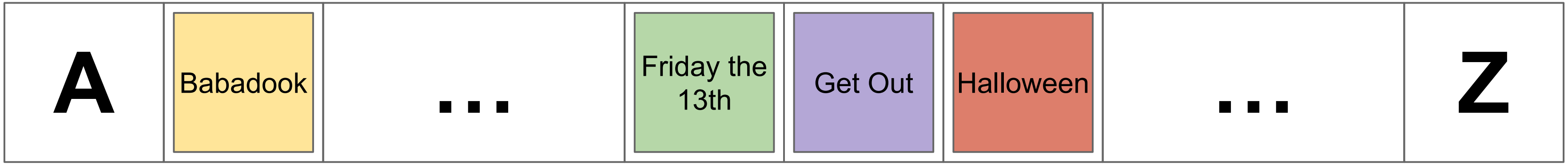
Halloween

...

Z

Assigning Bins

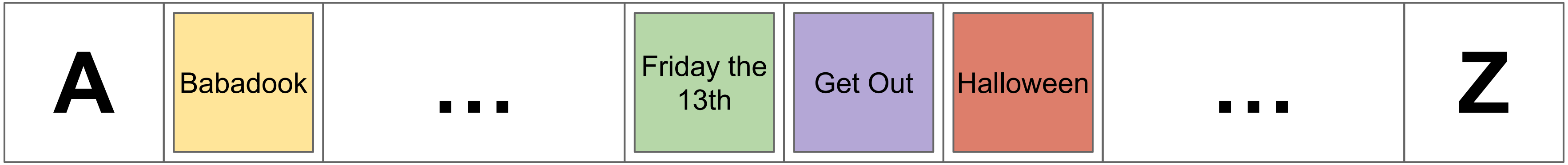
What about: `find("Hereditary")`?



A	Babadook	...	Friday the 13th	Get Out	Halloween	...	Z

Assigning Bins

What about: `find("Hereditary")`?
Once we know the location, we still need to check for an exact match.
"Hereditary"[0] == "H" == 7, Array[7] != "Hereditary"



A	Babadook	...	Friday the 13th	Get Out	Halloween	...	Z

Assigning Bins

Pros

- ***O***(1) insert
- ***O***(1) find
- ***O***(1) remove

Cons

- Wasted space (4/26 slots used in the example, will we ever use "Z"?)
- Duplication (What about inserting **F**rankenstein)

Bin-Based Organization

Wasted Space

- Not ideal…but not wrong
- ***O*(1)** access time might be worth it
- Also depends on the choice of function

Duplication

- We need to be able to handle duplicates

Bin-Based Organization

Wasted Space

- Not ideal…but not wrong
- ***O***(1) access time might be worth it
- Also depends on the choice of function

Duplication

- We need to be able to handle duplicates

What about "buckets" instead of "bins" (store multiple items per location)

Handling "Duplicates"

How can we store multiple items at each location?

Bigger Buckets

Fixed Size Buckets (*B* elements)

Pros

- Can deal with up to ***B*** dupes
- Still ***O***(1) find

Cons

- What if more than ***B*** dupes?

Arbitrarily Large Buckets (List)

Pros

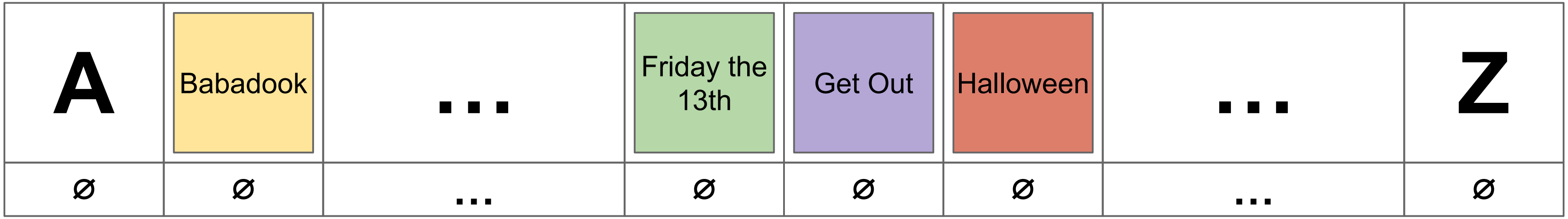
- No limit to number of dupes

Cons

- ***O***(***n***) worst-case find

Assigning Bins

`add("Frankenstein")?`



A	Babadook	...	Friday the 13th	Get Out	Halloween	...	Z
∅	∅	...	∅	∅	∅	...	∅

Assigning Bins

`add("Frankenstein")?`