CSE 250

Data Structures

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

Lec 30: Introduction to Hash Tables

Announcements

- Recitations **DO** meet this week
- Recitations **DO NOT** meet next week (Thanksgiving week)
- WA4 due Wednesday @ 11:59PM
- PA3 coming soon

Sets

A **Set** is an **unordered** collection of **unique** elements.

(order doesn't matter, and at most one copy of each item)

Sets

A **Set** is an **unordered** collection of **unique** elements.

(order doesn't matter, and at most one copy of each ~~item~~ key)

The Set ADT

`void add(T element)`

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

`boolean contains(T element)`

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

`boolean remove(T element)`

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

Implementing Sets/Bags

	`add`	`contains`	`remove`
ArrayList	$O(n)$	$O(n)$	$O(n)$
LinkedList	$O(n)$	$O(n)$	$O(n)$
Sorted ArrayList	$O(n)$	$O(\log(n))$	$O(n)$
Sorted LinkedList	$O(n)$	$O(n)$	$O(n)$

Implementing Sets/Bags

	`add`	`contains`	`remove`
ArrayList	$O(n)$	$O(n)$	$O(n)$
LinkedList	$O(n)$	$O(n)$	$O(n)$
Sorted ArrayList	$O(n)$	$O(\log(n))$	$O(n)$
Sorted LinkedList	$O(n)$	$O(n)$	$O(n)$
General BST	*??*	*??*	*??*
Balanced BST	*??*	*??*	*??*

Implementing Sets/Bags

	`add`	`contains`	`remove`
ArrayList	$O(n)$	$O(n)$	$O(n)$
LinkedList	$O(n)$	$O(n)$	$O(n)$
Sorted ArrayList	$O(n)$	$O(\log(n))$	$O(n)$
Sorted LinkedList	$O(n)$	$O(n)$	$O(n)$
General BST	$O(d) = O(n)$	$O(d) = O(n)$	$O(d) = O(n)$
Balanced BST	$O(d) = O(\log(n))$	$O(d) = O(\log(n))$	$O(d) = O(\log(n))$

Implementing Sets/Bags

	`add`	`contains`	`remove`
ArrayList	$O(n)$	$O(n)$	$O(n)$
LinkedList	$O(n)$	$O(n)$	$O(n)$
Sorted ArrayList	$O(n)$	$O(\log(n))$	$O(n)$
Sorted LinkedList			
General BST	$O(d) = O(n)$	$O(d) = O(n)$	$O(d) = O(n)$
Balanced BST	$O(d) = O(\log(n))$	$O(d) = O(\log(n))$	$O(d) = O(\log(n))$

Can we improve on this even further?

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**

contains => **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**

contains => **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, String →[0,26)

Assigning Bins

A	B	…	F	G	H	…	Z

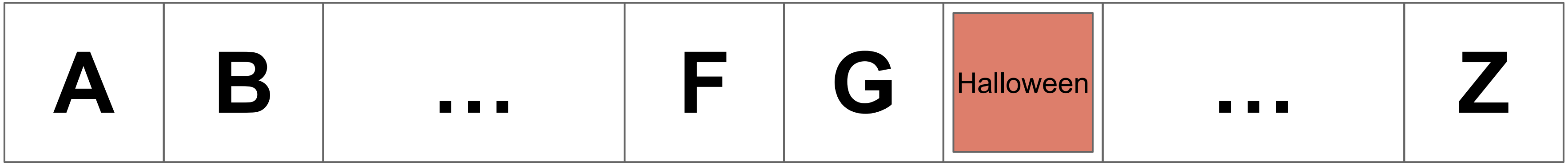
Assigning Bins

```
add("Halloween")
```

A	B	…	F	G	H	…	Z

Assigning Bins

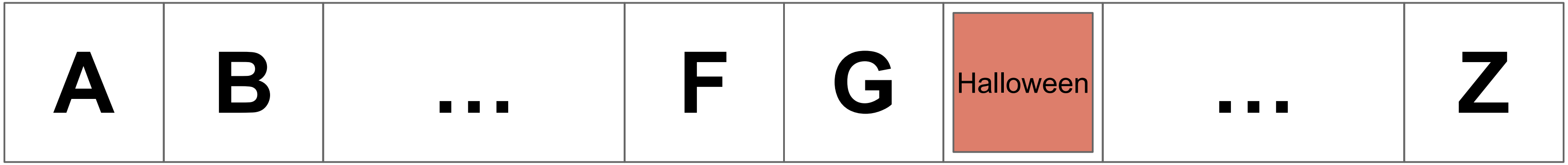
`add("Halloween") → "Halloween"[0] == "H" == 7`



Assigning Bins

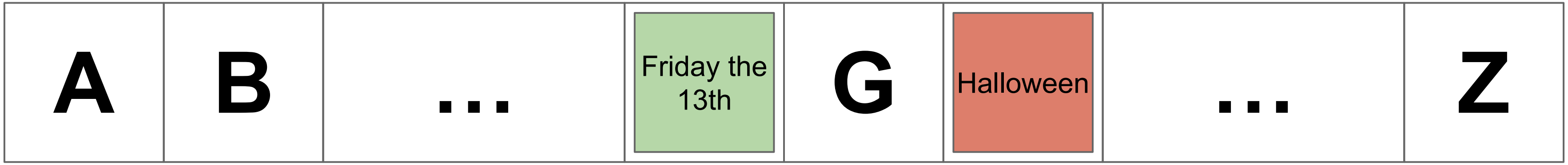
`add("Halloween")` → `"Halloween"[0] == "H" == 7`

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



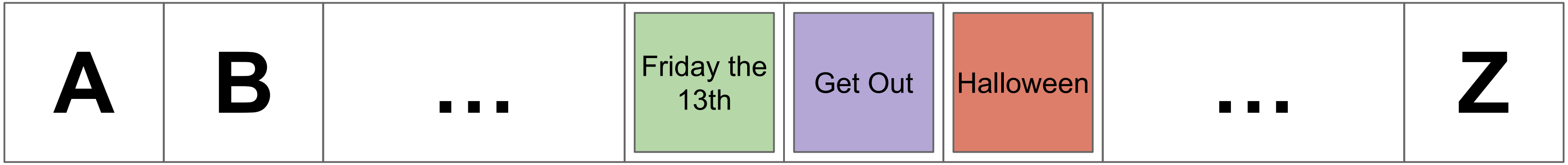
Assigning Bins

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



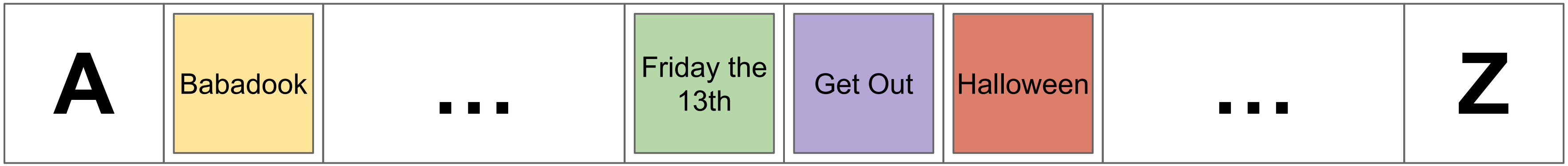
Assigning Bins

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



Assigning Bins

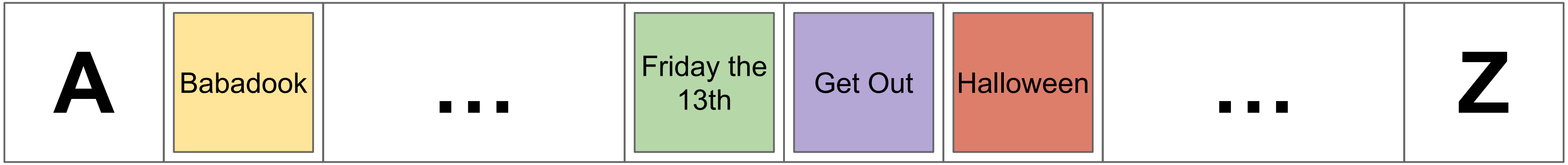
`add("Babadook") → "Babadook"[0] == "B" == 1`



Assigning Bins

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

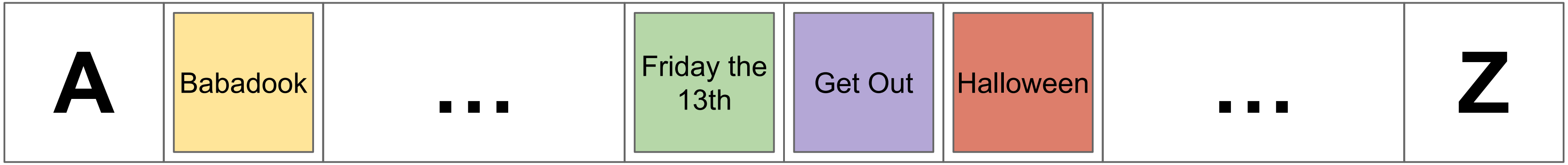
Find in constant time!



Assigning Bins

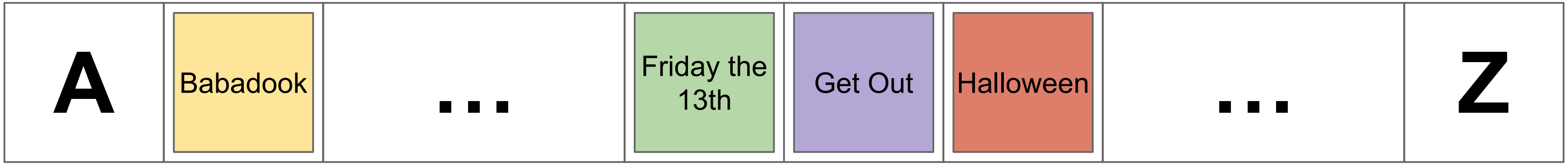
`contains("Scream")` → `"Scream"[0] == "S" == 18`

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



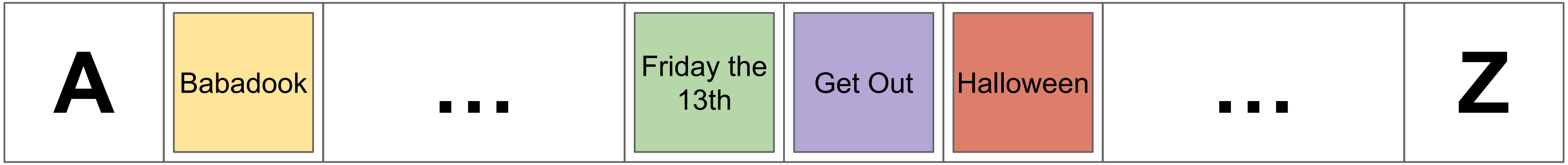
Assigning Bins

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



Assigning Bins

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



Once we know the location, we still need to check for an exact match.

`"Hereditary"[0] == "H" == 7, Array[7] != "Hereditary"`

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

Assigning Bins

Pros (so far...)

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

Cons?

Assigning Bins

Pros (so far...)

- ***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 hash 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 hash function

Duplication

- We need to be able to handle duplicates!

What about "buckets" that can store multiple items?

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

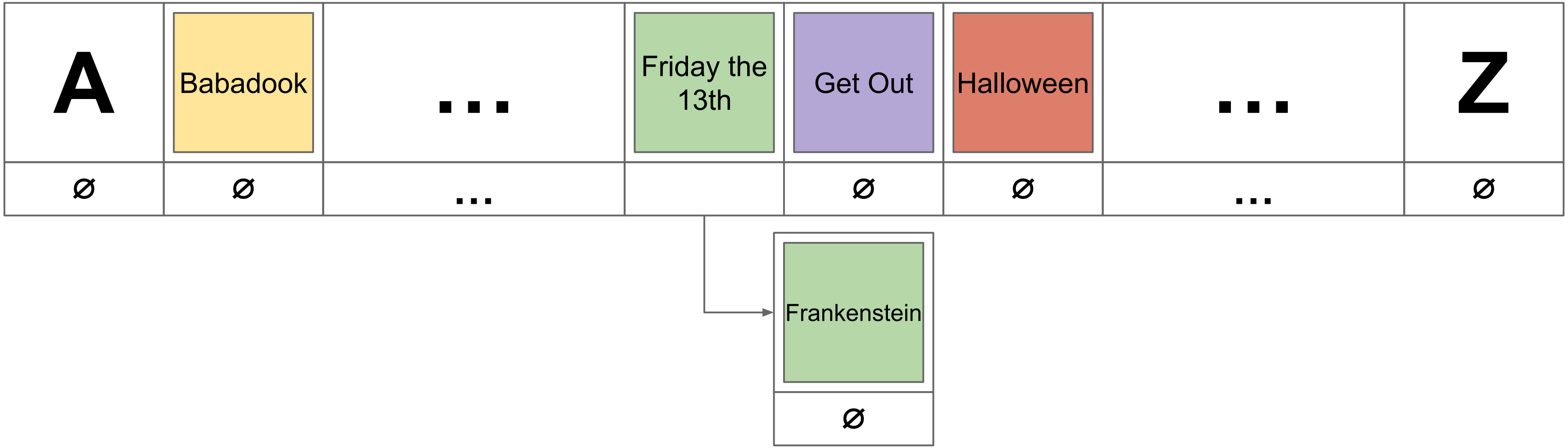
Assigning Bins

`add("Frankenstein")?`

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

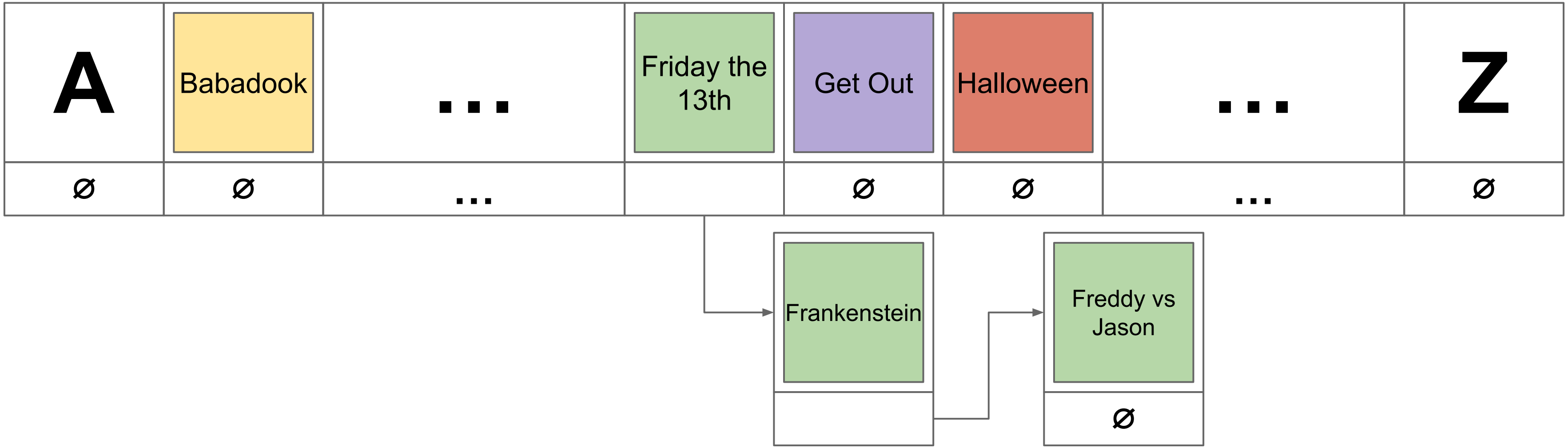
Assigning Bins

`add("Frankenstein")?`



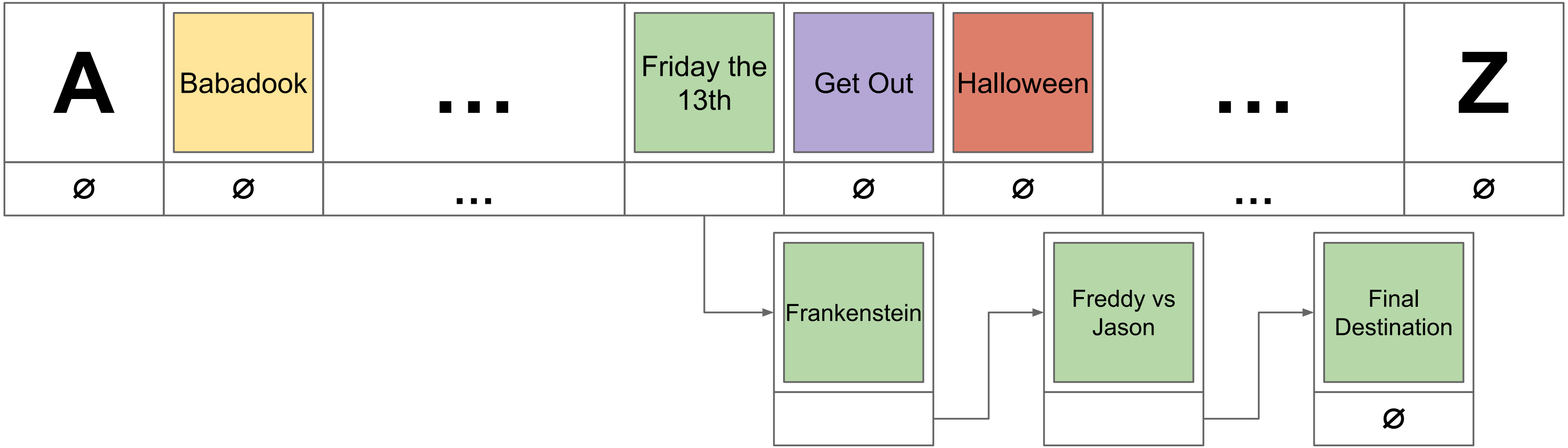
Assigning Bins

`add("Freddy vs Jason")?`



Assigning Bins

```
add("Final Destination")?
```



LinkedList Bins

Now we can handle as many duplicates as we need. But are we losing our constant time operations?

How many elements are we expecting to end up in each bucket?

LinkedList Bins

Now we can handle as many duplicates as we need. But are we losing our constant time operations?

How many elements are we expecting to end up in each bucket?

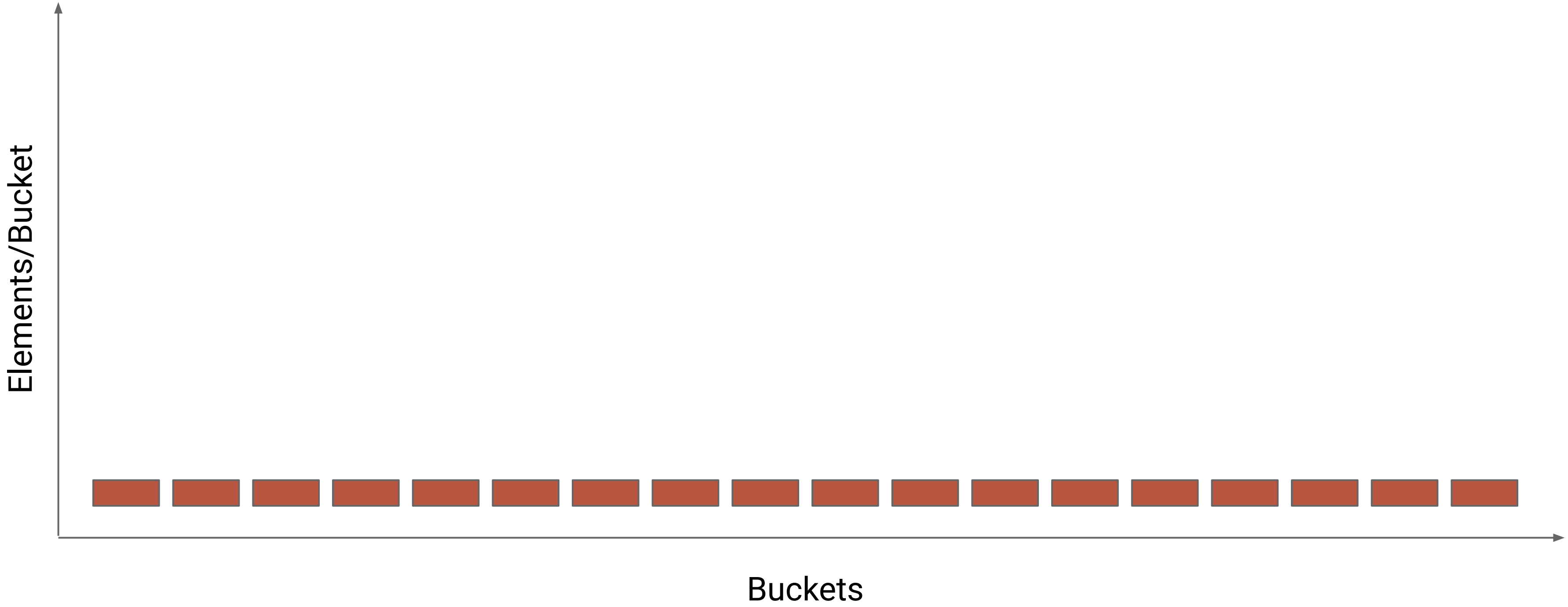
Depends partially on our choice of Hash Function

Picking a Hash Function

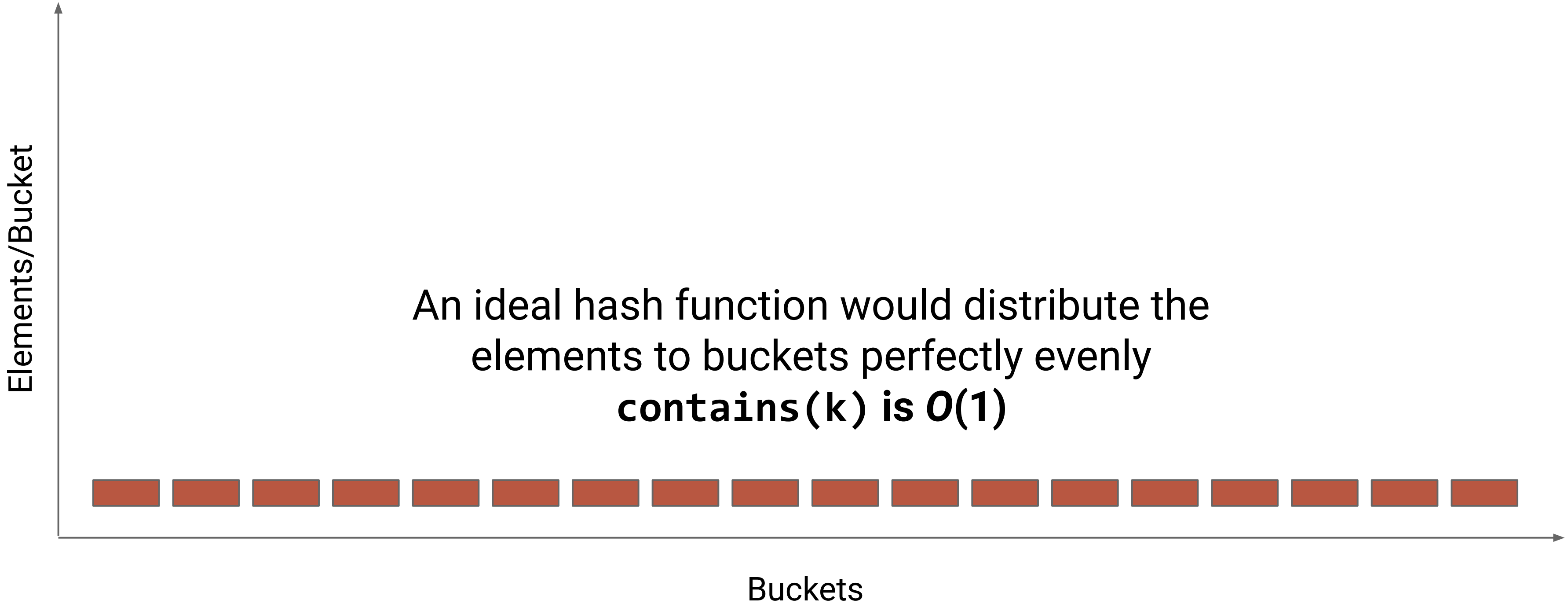
Desirable features for *h*(*x*):

- Fast — needs to be ***O*(1)**
- "Unique" – As few duplicate bins as possible

Picking a Hash Function



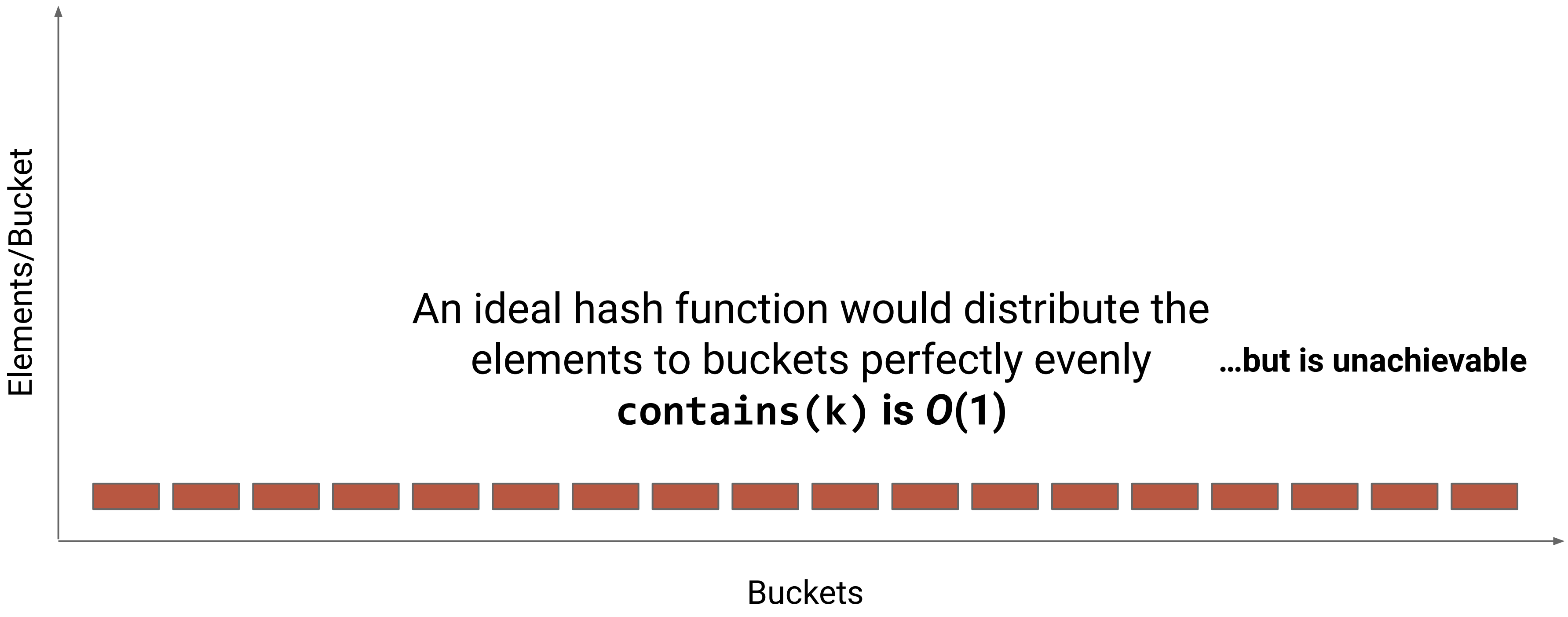
Picking a Hash Function



An ideal hash function would distribute the elements to buckets perfectly evenly

`contains(k)` is *O*(1)

Picking a Hash Function

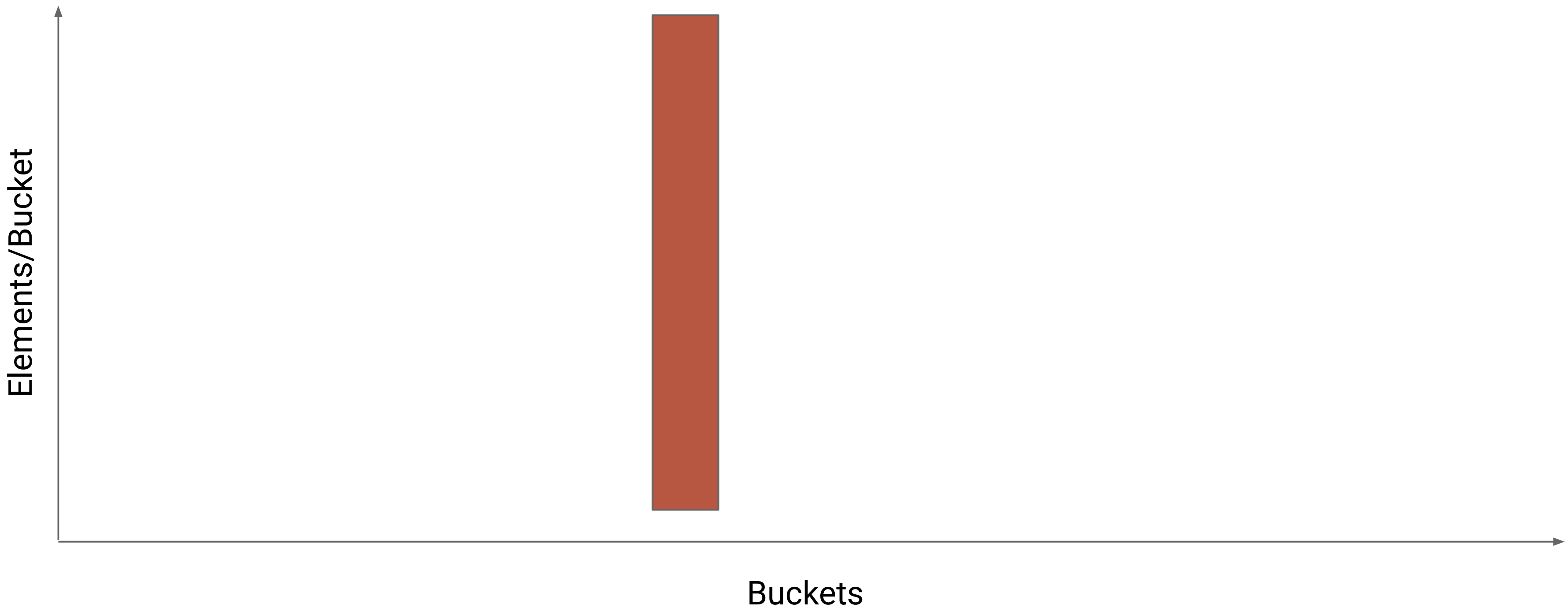


An ideal hash function would distribute the elements to buckets perfectly evenly

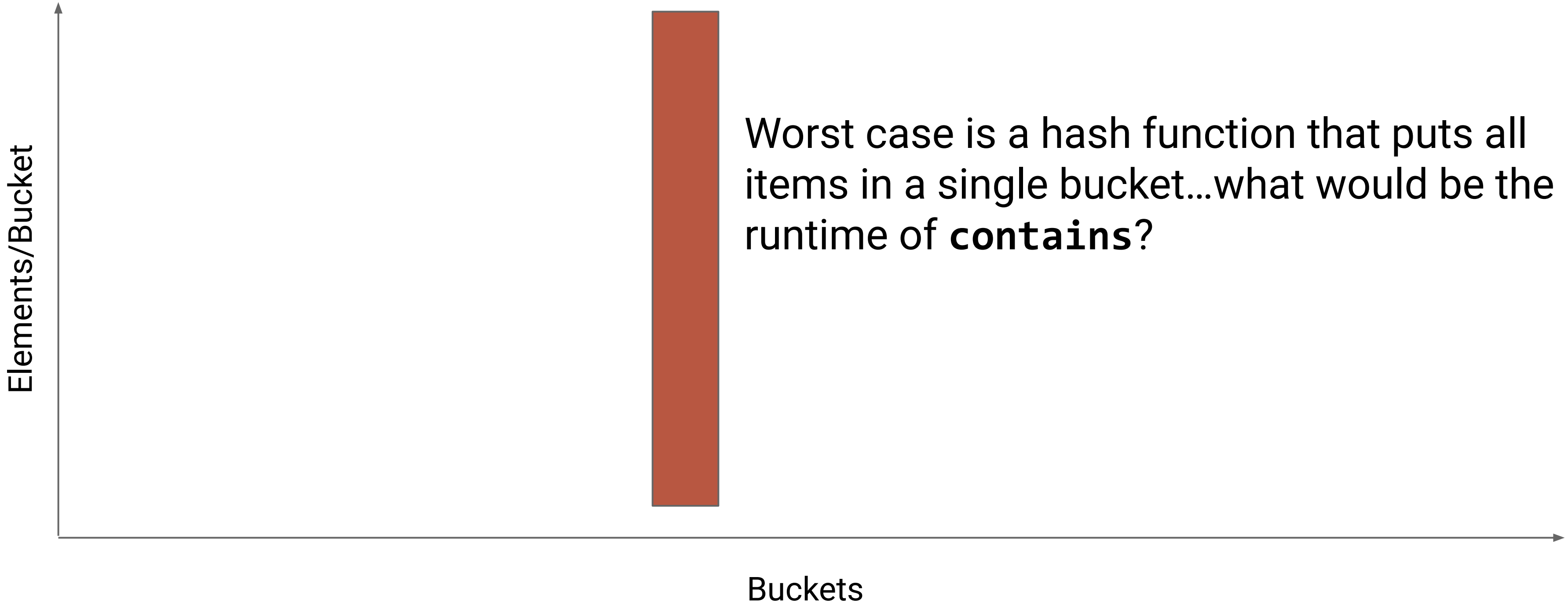
`contains(k)` is *O*(1)

…but is unachievable

Picking a Hash Function

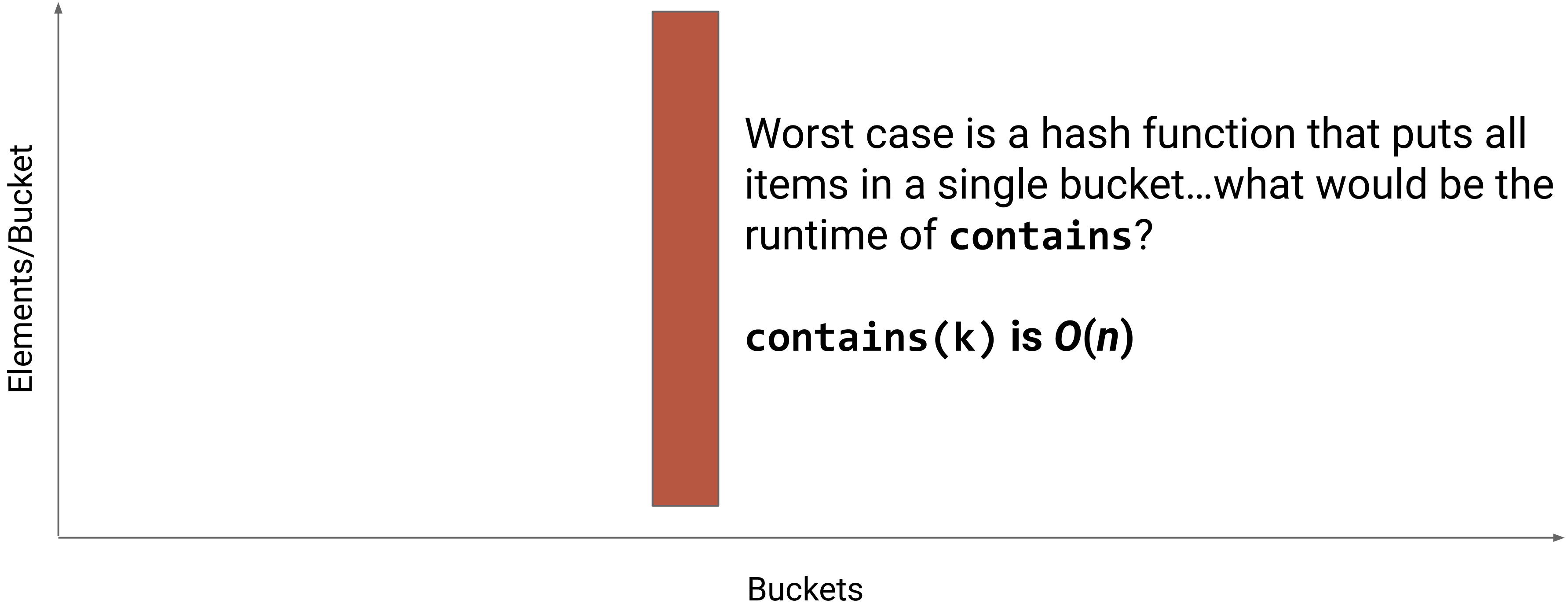


Picking a Hash Function



Worst case is a hash function that puts all items in a single bucket…what would be the runtime of **`contains`**?

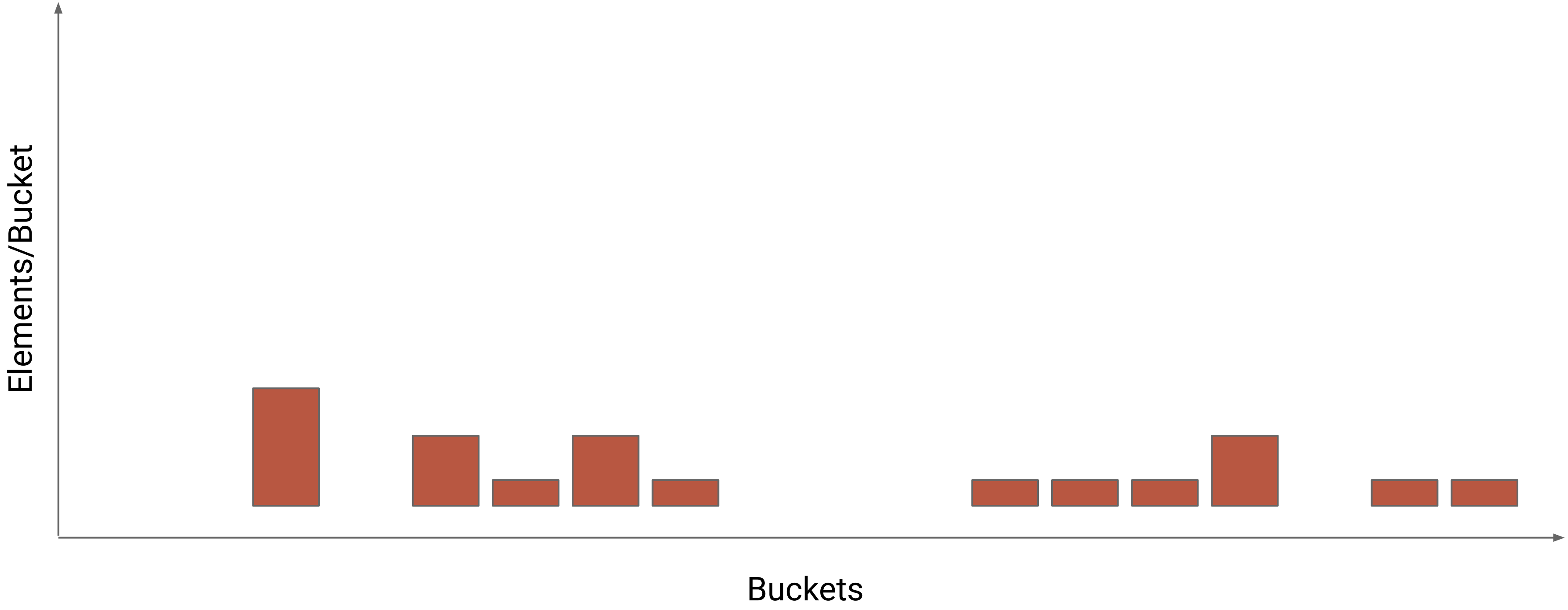
Picking a Hash Function



Worst case is a hash function that puts all items in a single bucket…what would be the runtime of **`contains`**?

`contains(k)` is *O*(*n*)

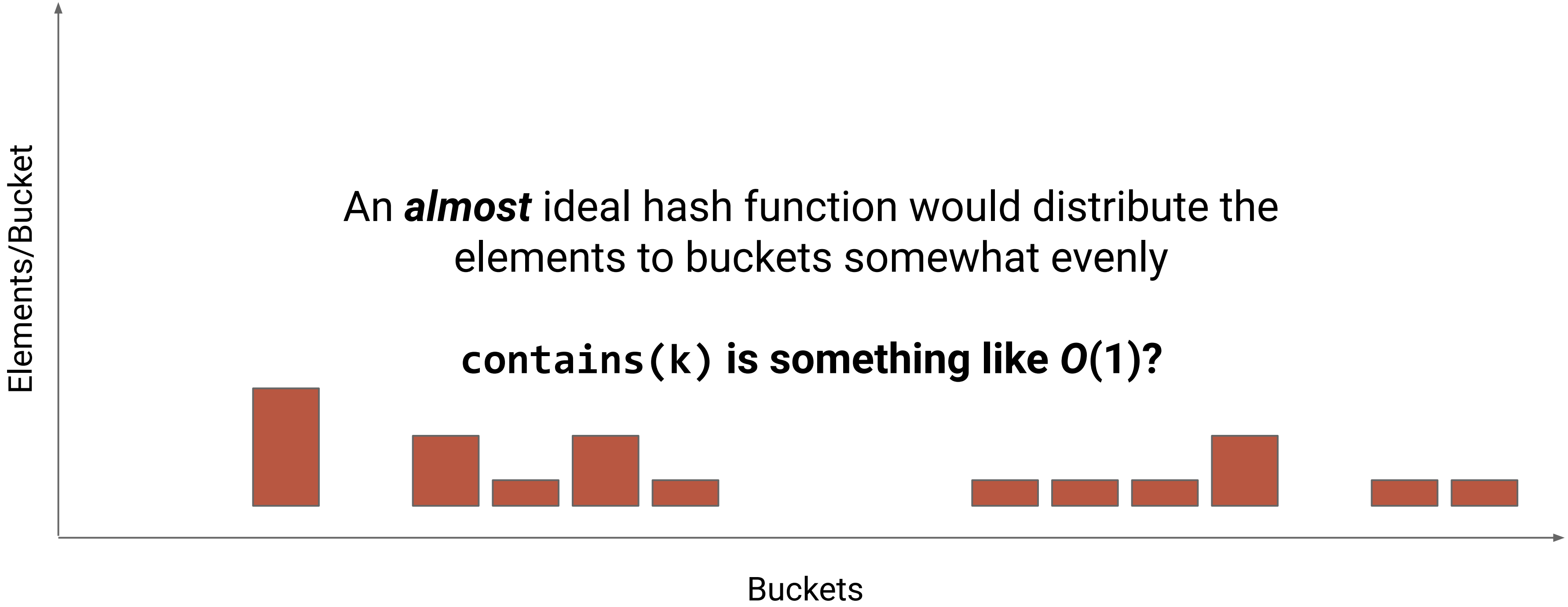
Picking a Hash Function



Picking a Hash Function

An ***almost*** ideal hash function would distribute the elements to buckets somewhat evenly

`contains(k)` is something like *O*(1)?

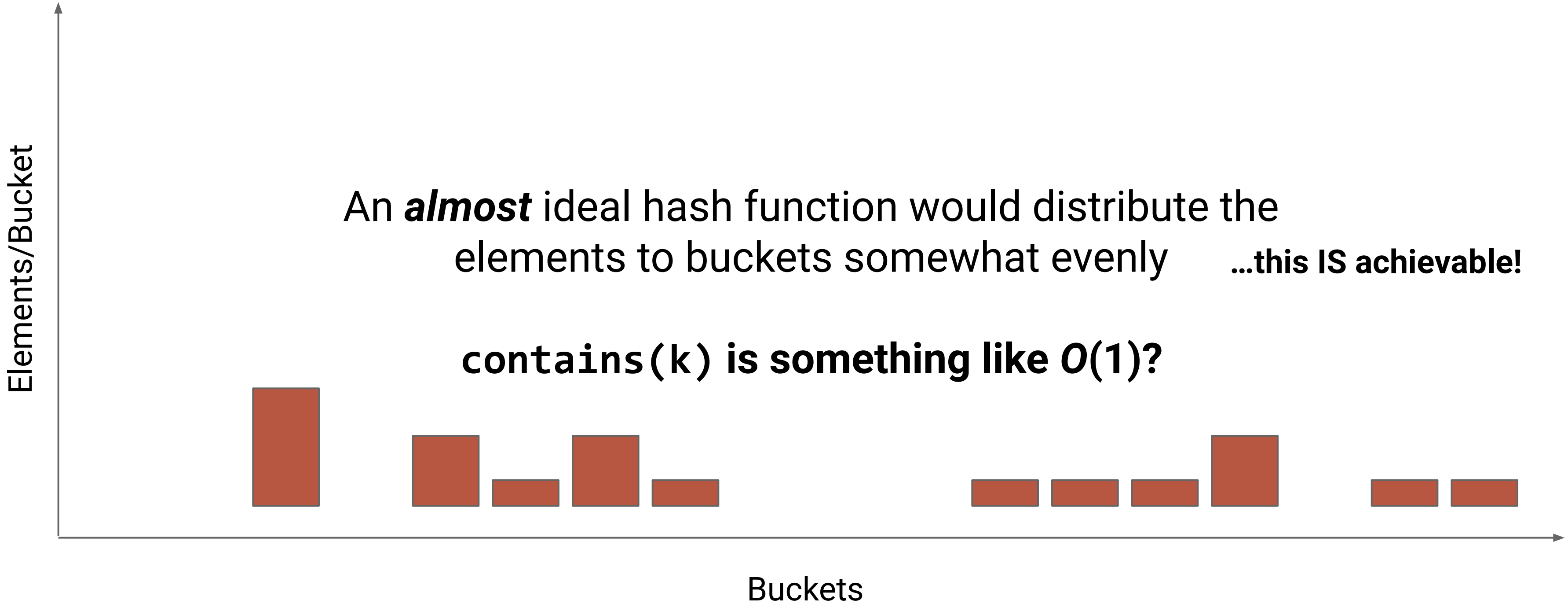


Picking a Hash Function

An ***almost*** ideal hash function would distribute the elements to buckets somewhat evenly

...this IS achievable!

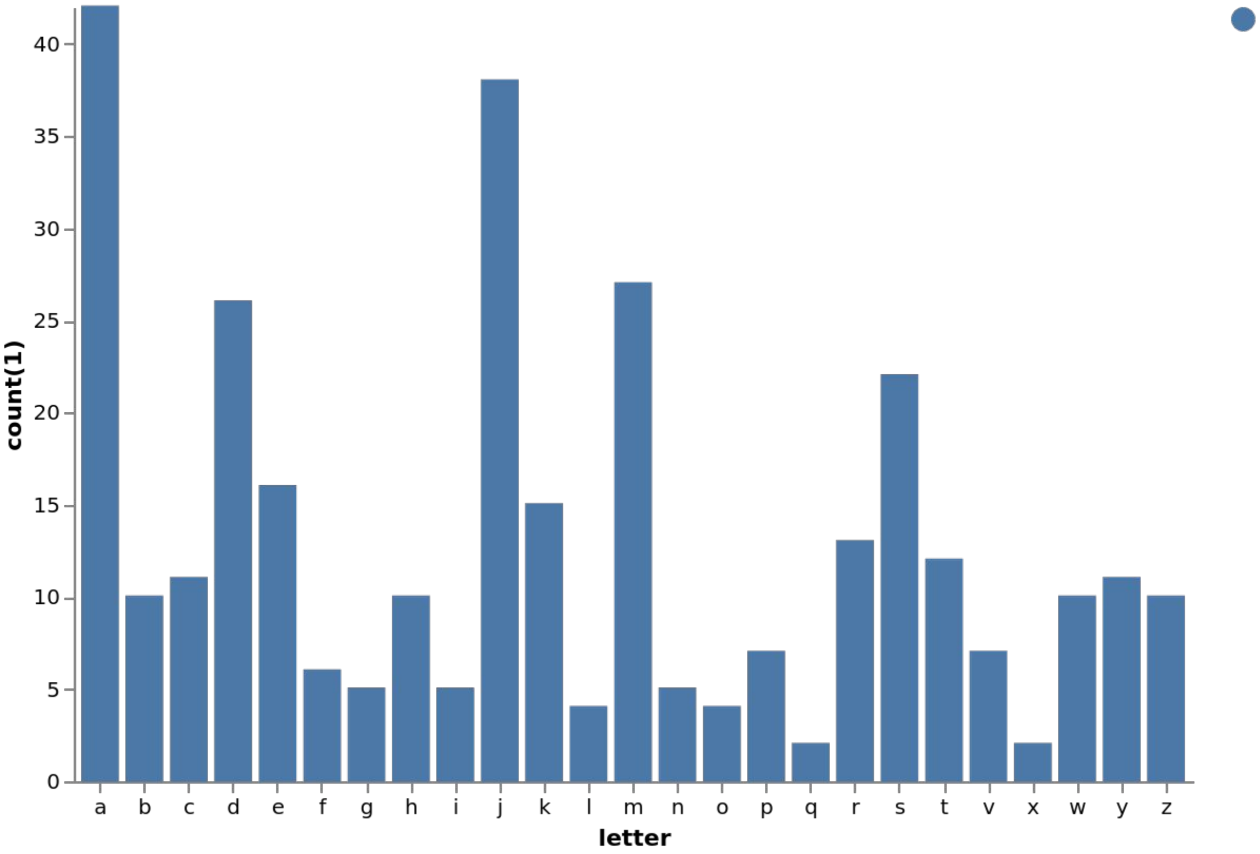
`contains(k)` is something like *O*(1)?



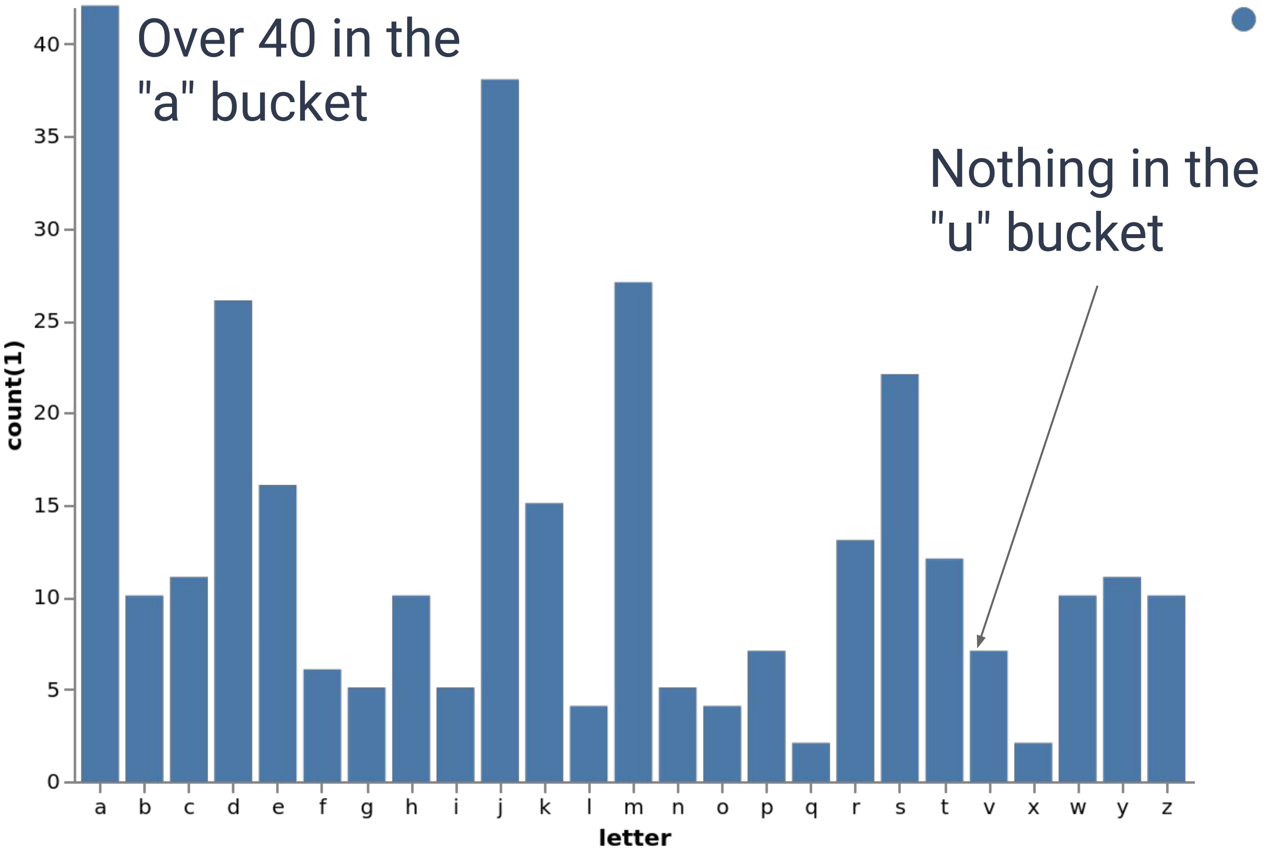
Example Hash Functions

First Letter of UBIT Name

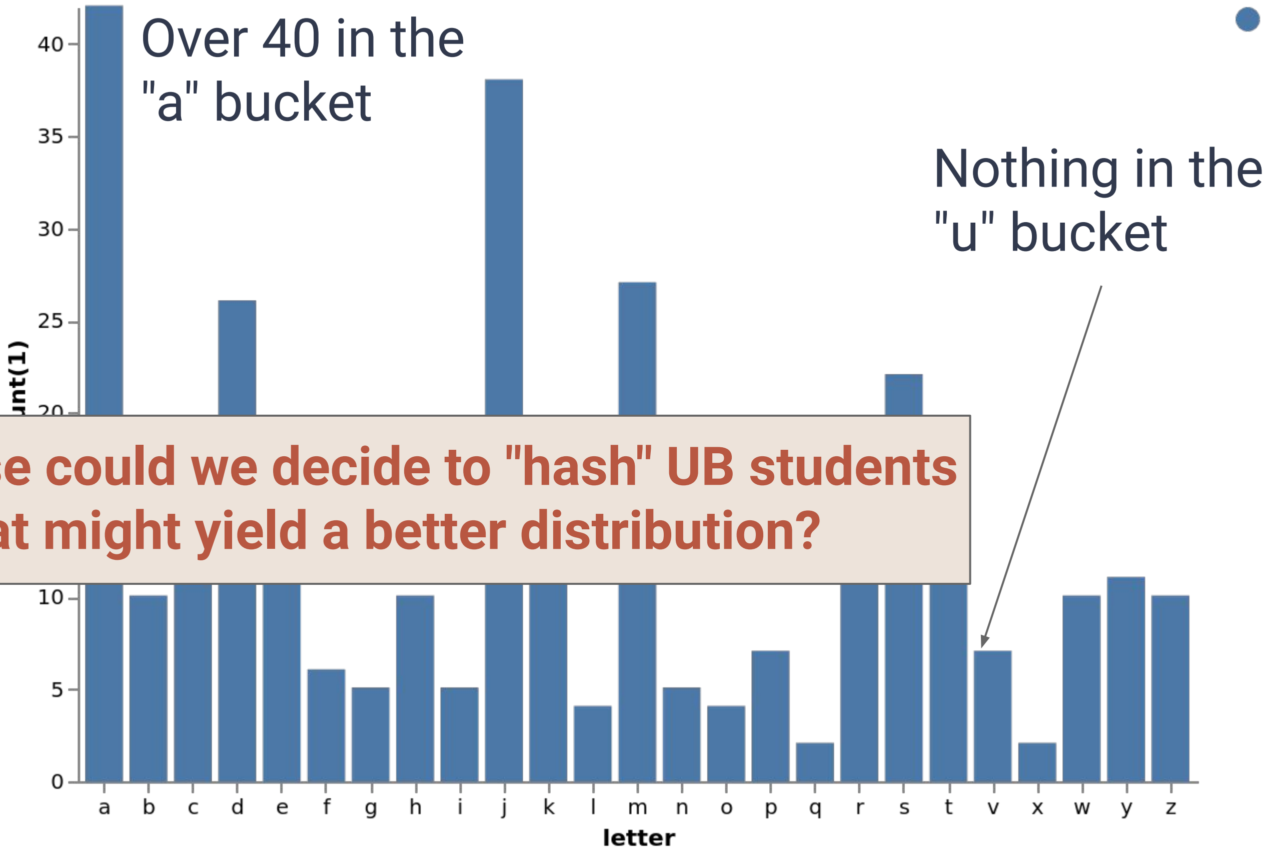
- Unevenly distributed, $O(n)$ worst case apply



Distribution of UBIT Names to Buckets based on first letter



Distribution of UBIT Names to Buckets based on first letter



How else could we decide to "hash" UB students that might yield a better distribution?

Distribution of UBIT Names to Buckets based on first letter

Other Functions

First Letter of UBIT Name

- Unevenly distributed, $\boldsymbol{O(n)}$ worst case apply

Identity Function on UBIT #

- Need a 50m+ element array

Other Functions

First Letter of UBIT Name

- Unevenly distributed, $O(n)$ worst case apply

Identity Function on UBIT #

- Need a N = 50m+ element array
- **Problem:** For reasonable N, identity function returns something > N

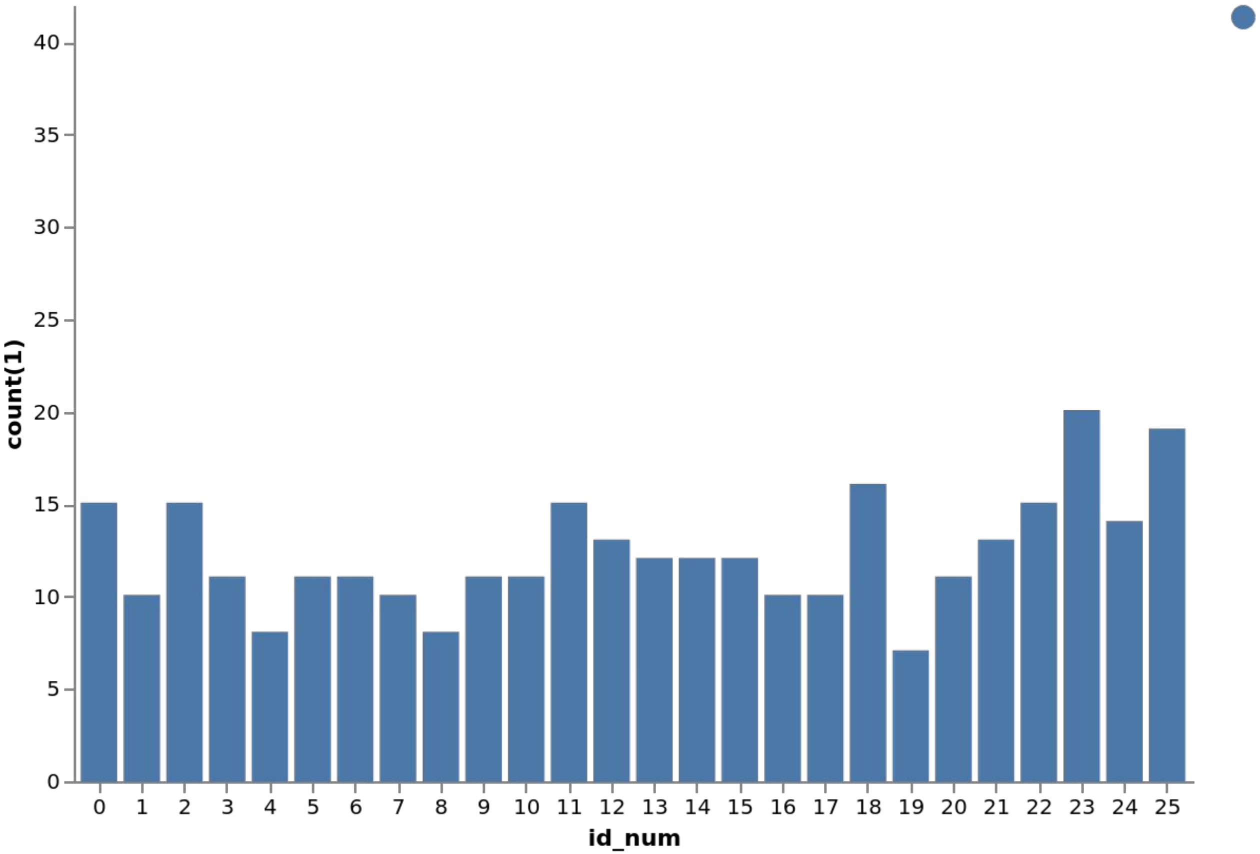
Other Functions

First Letter of UBIT Name

- Unevenly distributed, ***O*(*n*)** worst case apply

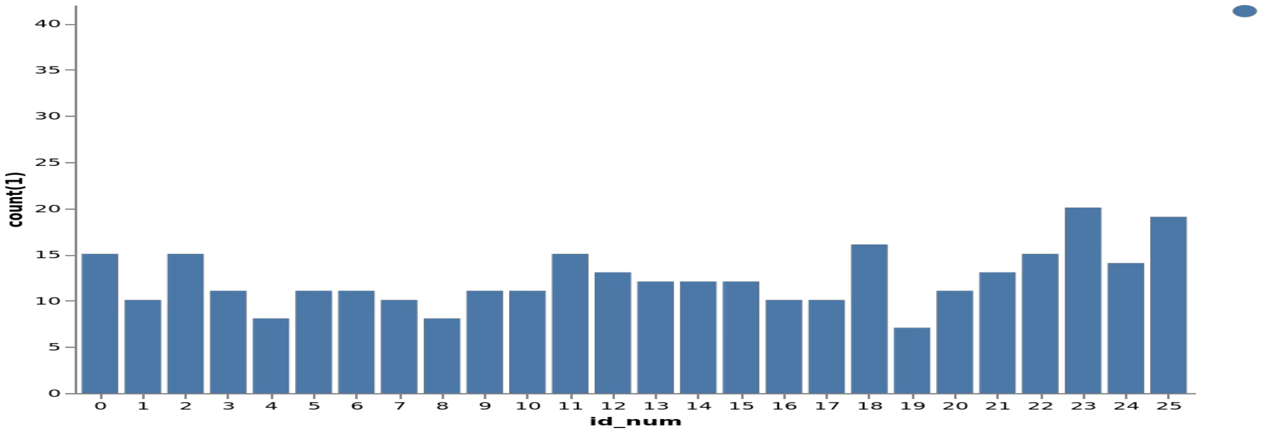
Identity Function on UBIT #

- Need a ***N*** = 50m+ element array
- **Problem:** For reasonable ***N*****,** identity function returns something > ***N***
- **Solution:** Cap return value of function to ***N*** with modulus
  - `return h(x) % N`

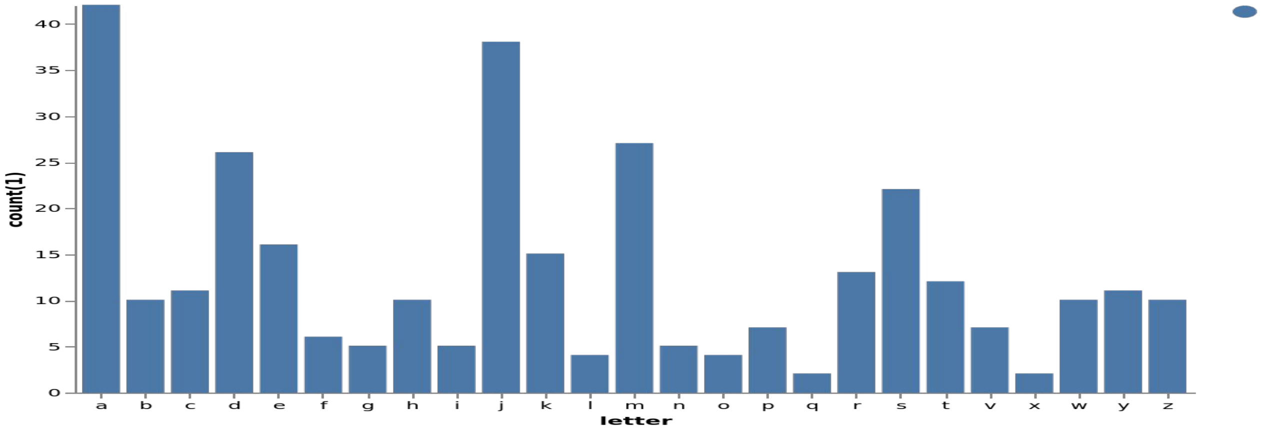


Distribution of Person # % 26

Person # % 26
More even distribution



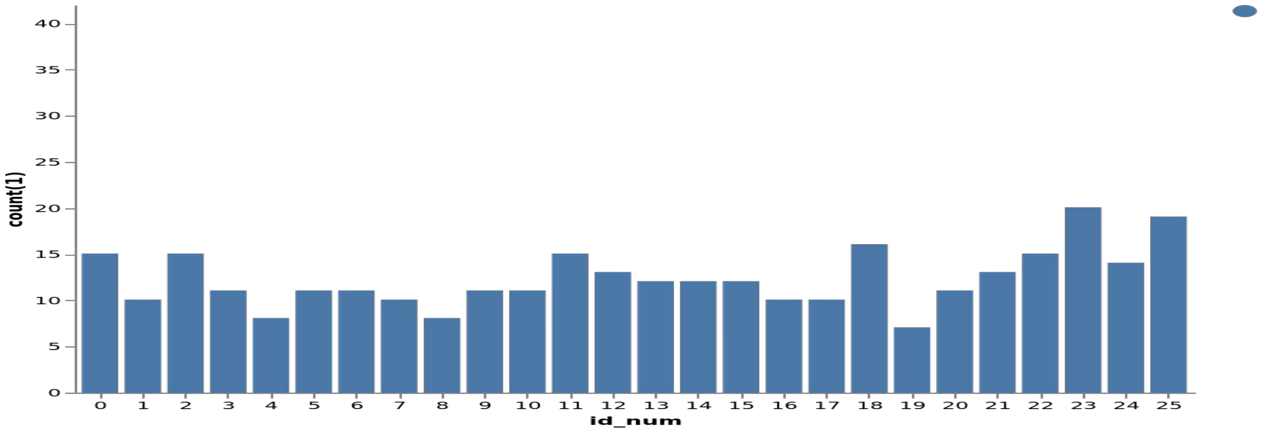
First letter of UBIT name



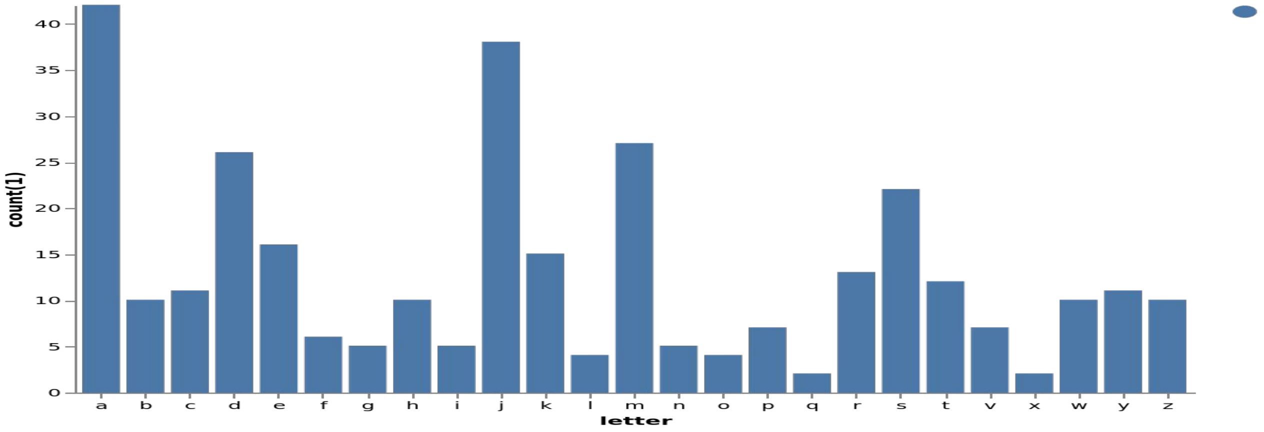
Hash Function Comparison

Person # % 26
More even distribution

(does rely on Person #s being somewhat "randomly" distributed)



First letter of UBIT name



Hash Function Comparison

Picking a Hash Function

What else could we use that would evenly distribute values to locations?

Picking a Hash Function

What else could we use that would evenly distribute values to locations?

Wacky Idea: Have ***h***(***x***) return a random value in **[0,N)**

(This makes ***`contains`*** *impossible…but bear with me)*

Random Hash Function

$$n = \text{number of elements in any bucket}$$

$$N = \text{number of buckets}$$

$$b_{i,j} = \begin{cases} 1 & \textbf{if } \text{element } i \text{ is assigned to bucket } j \\ 0 & otherwise \end{cases}$$

Random Hash Function

$$n = \text{number of elements in any bucket}$$

$$N = \text{number of buckets}$$

$$b_{i,j} = \begin{cases} 1 & \textbf{if } \text{element } i \text{ is assigned to bucket } j \\ 0 & otherwise \end{cases}$$

$$\mathbb{E}\left[b_{i,j}\right] = \frac{1}{N}$$

Random Hash Function

$$n = \text{number of elements in any bucket}$$

$$N = \text{number of buckets}$$

$$b_{i,j} = \begin{cases} 1 & \textbf{if } \text{element } i \text{ is assigned to bucket } j \\ 0 & otherwise \end{cases}$$

$$\mathbb{E}\left[\sum_{i=0}^{n} b_{i,j}\right] = \frac{n}{N}$$

Random Hash Function

$$n = \text{number of elements in any bucket}$$

$$N = \text{number of buckets}$$

$$b_{i,j} = \begin{cases} 1 & \textbf{if } \text{element } i \text{ is assigned to bucket } j \\ 0 & otherwise \end{cases}$$

$$\mathbb{E}\left[\sum_{i=0}^{n} b_{i,j}\right] = \frac{n}{N}$$

Only true if $b_{i,j}$ and $b_{i',j}$ are uncorrelated for any i ≠ i'

(h(i) can't be related to h(i'))

The **expected** number of elements in any bucket j

Random Hash Function

$$n = \text{number of elements in any bucket}$$

$$N = \text{number of buckets}$$

$$b_{i,j} = \begin{cases} 1 & \textbf{if } \text{element } i \text{ is assigned to bucket } j \\ 0 & \textit{otherwise} \end{cases}$$

Only true if $b_{i,j}$ and $b_{i',j}$ are uncorrelated for any $i \neq i'$

(h(i) can't be related to h(i'))

$$\mathbb{E}\left[\sum_{i=0}^{n} b_{i,j}\right] = \frac{n}{N}$$

The **expected** number of elements in any bucket j

…given this information, what do the runtimes of our operations look like?

Random Hash Function

$$n = \text{number of elements in any bucket}$$

$$N = \text{number of buckets}$$

$$b_{i,j} = \begin{cases} 1 & \textbf{if } \text{element } i \text{ is assigned to bucket } j \\ 0 & otherwise \end{cases}$$

Expected runtime of **`insert`**, **`apply`**, **`remove`**: ***O*(*n*/*N*)**

Worst-Case runtime of **`insert`**, **`apply`**, **`remove`**: ***O*(*n*)**

Hash Functions In the Real-World

Examples

- SHA256 ← Used by GIT
- MD5, BCRYPT ← Used by unix login, apt
- MurmurHash3 ← Used by Scala

hash(*x*) is pseudo-random

- **hash(*x*)** ~ uniform random value in [0, INT_MAX)
- **hash(*x*)** always returns the same value for the same ***x***
- **hash(*x*)** is uncorrelated with **hash(*y*)** for all x ≠ y

Hash Functions + Buckets

Everything is: $O\left(\frac{n}{N}\right)$

Let's call $\alpha = \frac{n}{N}$ the load factor.

Hash Functions + Buckets

Everything is: $O\left(\frac{n}{N}\right)$

Let's call $\alpha = \frac{n}{N}$ the load factor.

Idea: Make α a constant

Hash Functions + Buckets

Everything is: $O\left(\frac{n}{N}\right)$ Let's call $\alpha = \frac{n}{N}$ the load factor.

Idea: Make α a constant

Fix an α_{max} and start requiring that $\alpha \leq \alpha_{max}$

Hash Functions + Buckets

Everything is: $O\left(\frac{n}{N}\right)$

Let's call $\alpha = \frac{n}{N}$ the load factor.

Idea: Make α a constant

Fix an α_{max} and start requiring that $\alpha \leq \alpha_{max}$

What do we do when this constraint is violated?

Hash Functions + Buckets

Everything is: $O\left(\frac{n}{N}\right)$

Let's call $\alpha = \frac{n}{N}$ the load factor.

Idea: Make α a constant

Fix an α_{max} and start requiring that $\alpha \leq \alpha_{max}$

What do we do when this constraint is violated? **Resize!**