CSE 250
Data Structures

Dr. Eric Mikida
epmikida@buffalo.edu

Dr. Oliver Kennedy
okennedy@buffalo.edu

212 Capen Hall

Day 29
Hash Functions

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

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` $O(\log(n))$

Finding Items

For most of these operations, the expensive part is **finding** the record…

Finding Items

For most of these operations, the expensive part is **finding** the record…

So…let's skip the search

Assigning Bins

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: If our records are names, first letter of name → [0,26)

Assigning Bins

A	B	C	D	...	P	...	Z

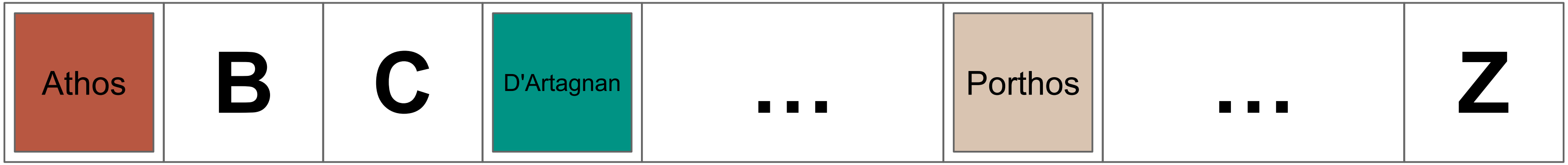
Assigning Bins

Athos	B	C	D	...	P	...	Z

Assigning Bins

Athos	B	C	D	...	Porthos	...	Z

Assigning Bins



Assigning Bins

Pros

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

Cons

- Wasted space (3/26 slots used in the example)
- Duplication (What about inserting **A**ramis)

Assigning ~~Bins~~ Buckets

Pros

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

Cons

- Wasted space (3/26 slots used in the example)
- Duplication (What about inserting **A**ramis)

Bucket-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 duplcates

Dealing with Duplication

How could we address the duplication problem?

Dealing with Duplication

How could we address the duplication problem?

Idea: Make buckets bigger!

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

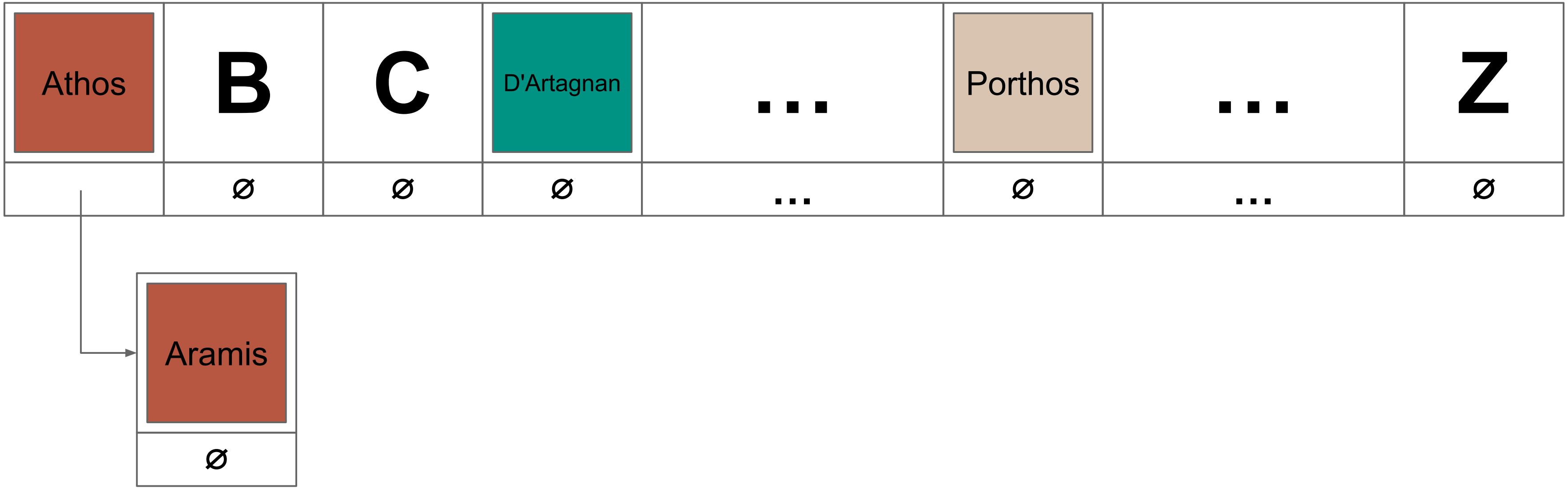
Cons

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

Buckets + Linked Lists

Athos	B	C	D'Artagnan	...	Porthos	...	Z
∅	∅	∅	∅	...	∅	...	∅

Buckets + Linked Lists

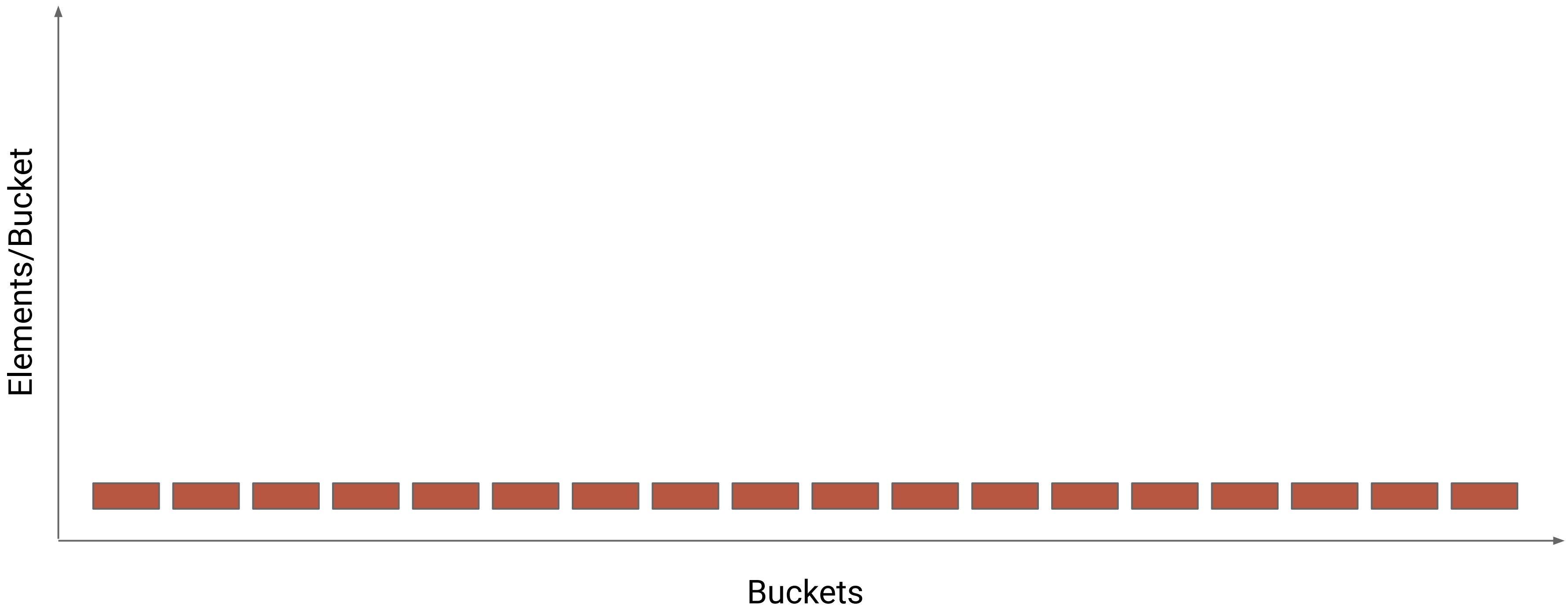


Picking a Hash Function

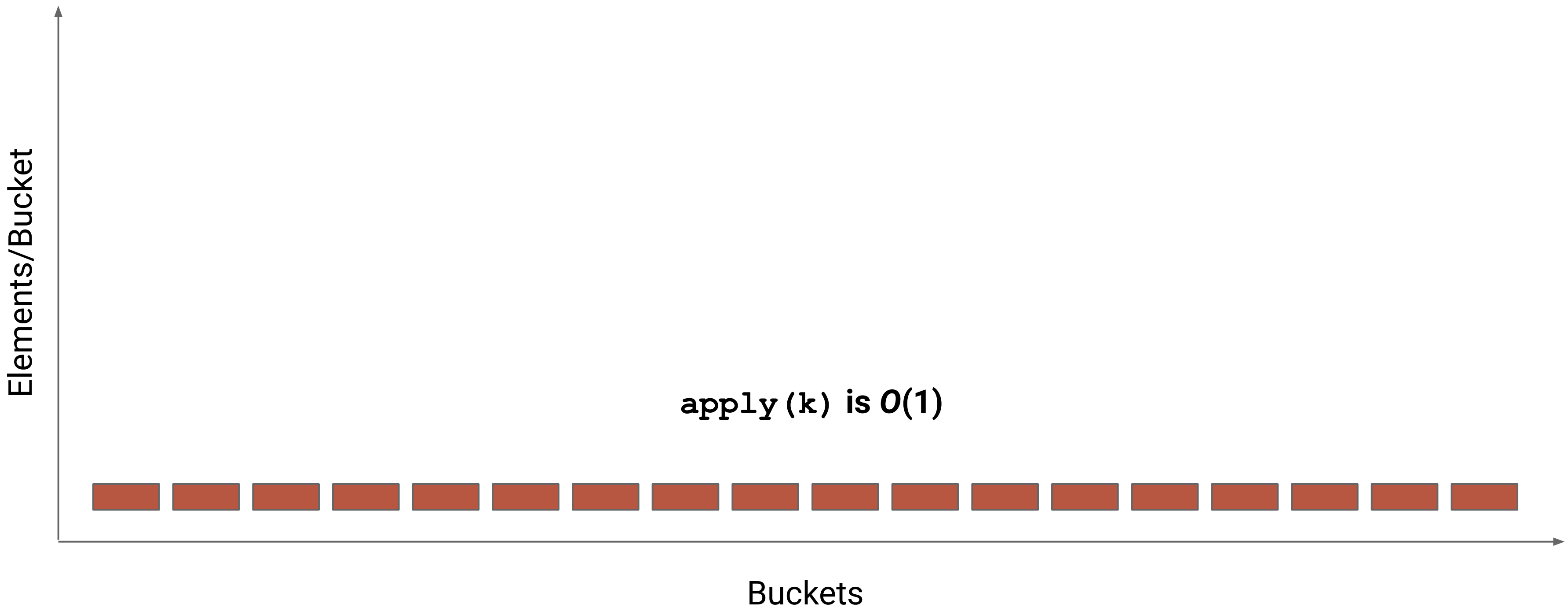
Desirable features for *h*(*x*):

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

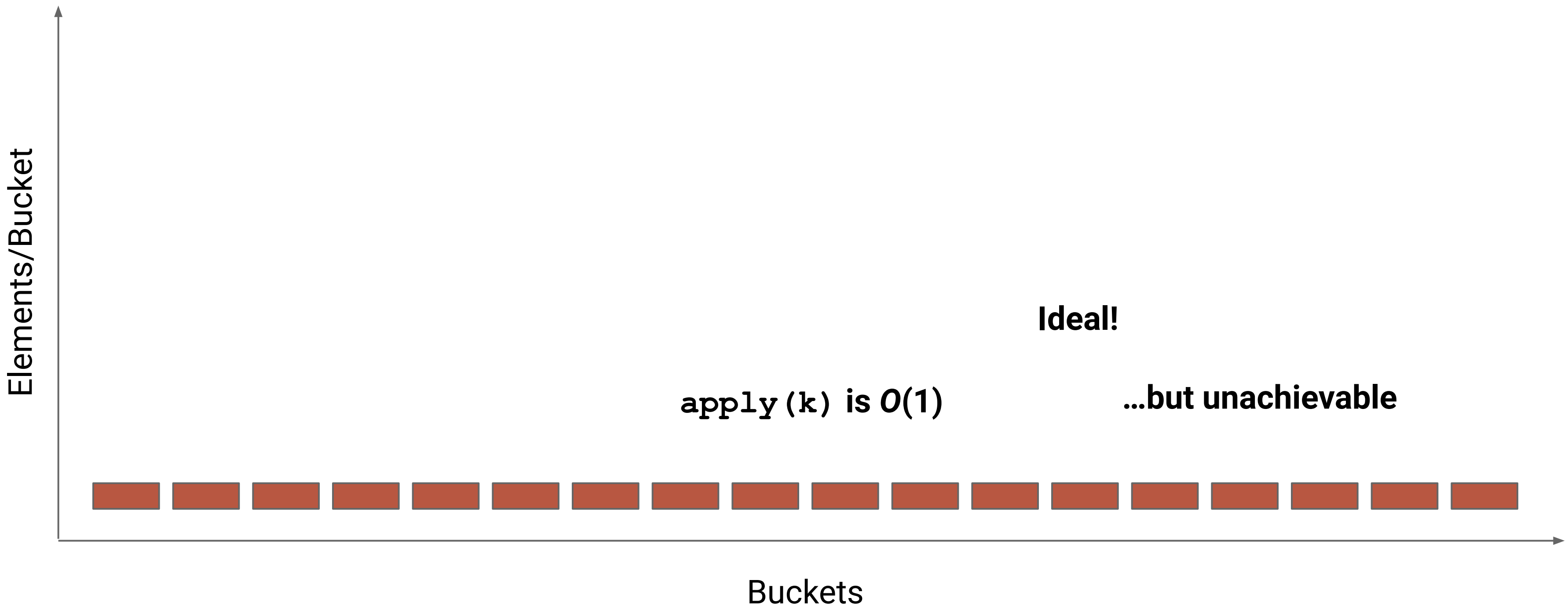
Picking a Hash Function



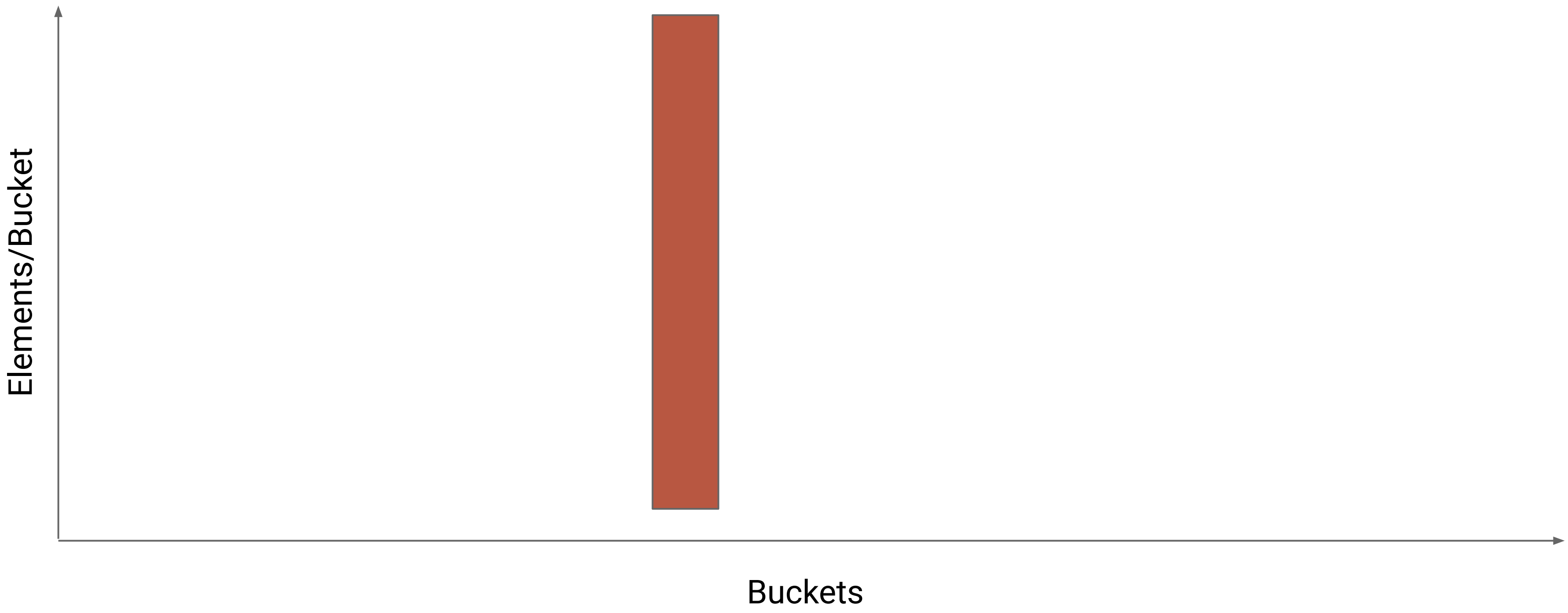
Picking a Hash Function



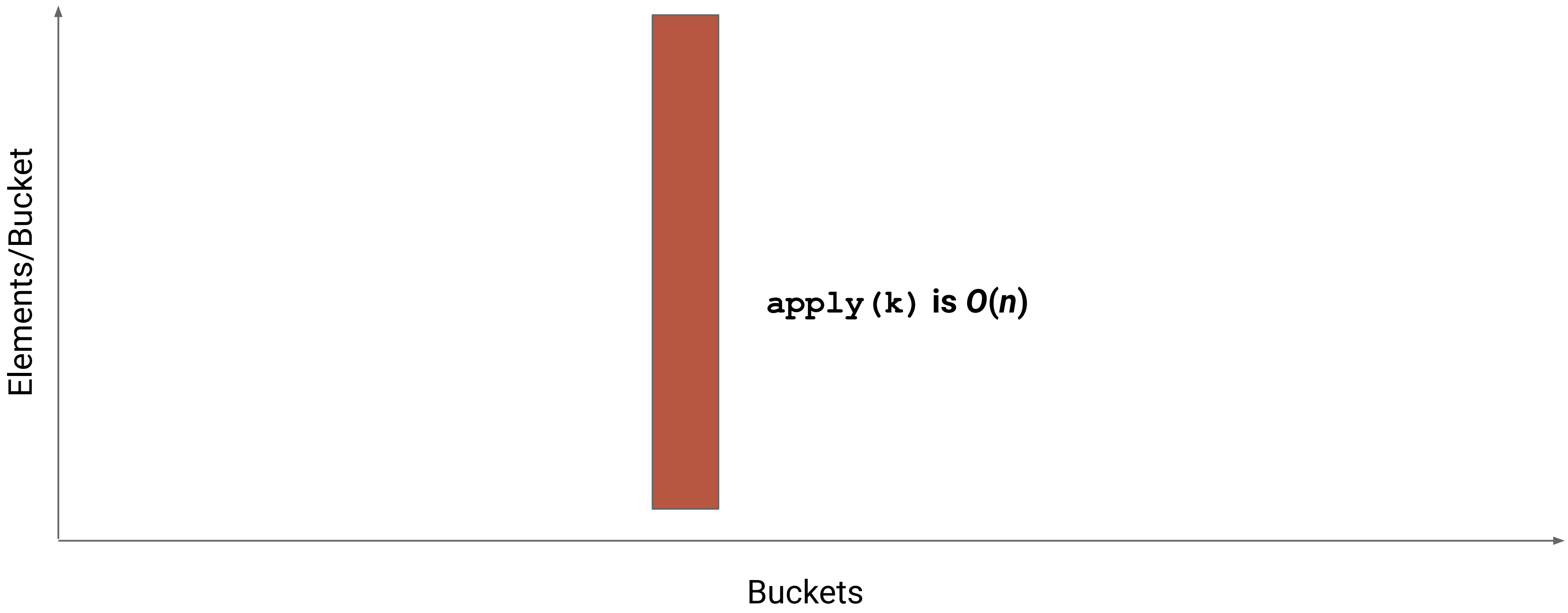
Picking a Hash Function



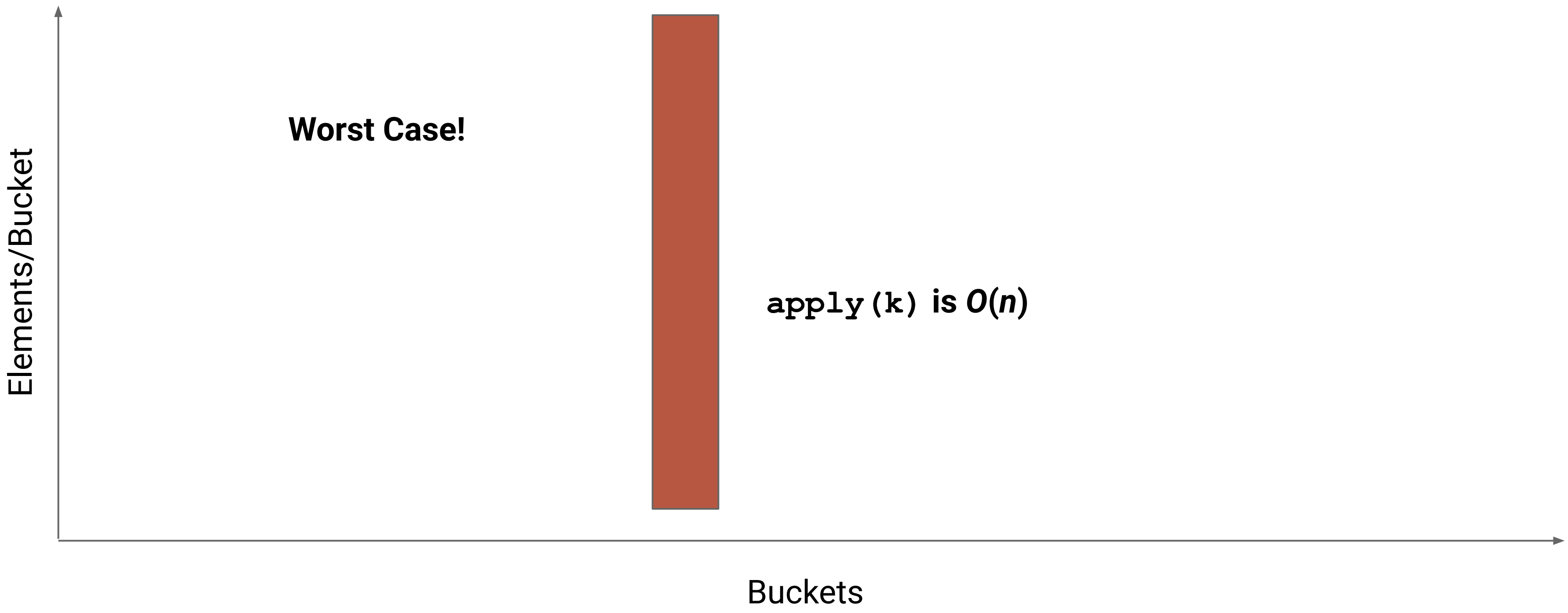
Picking a Hash Function



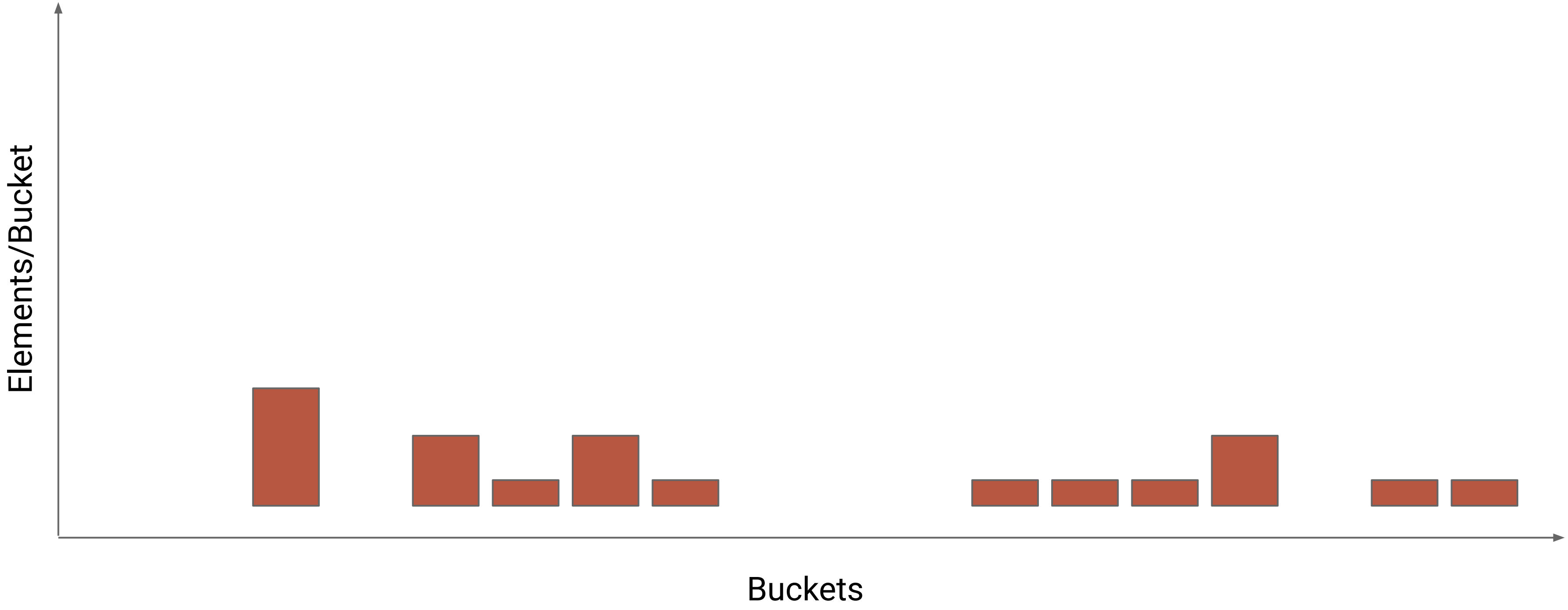
Picking a Hash Function



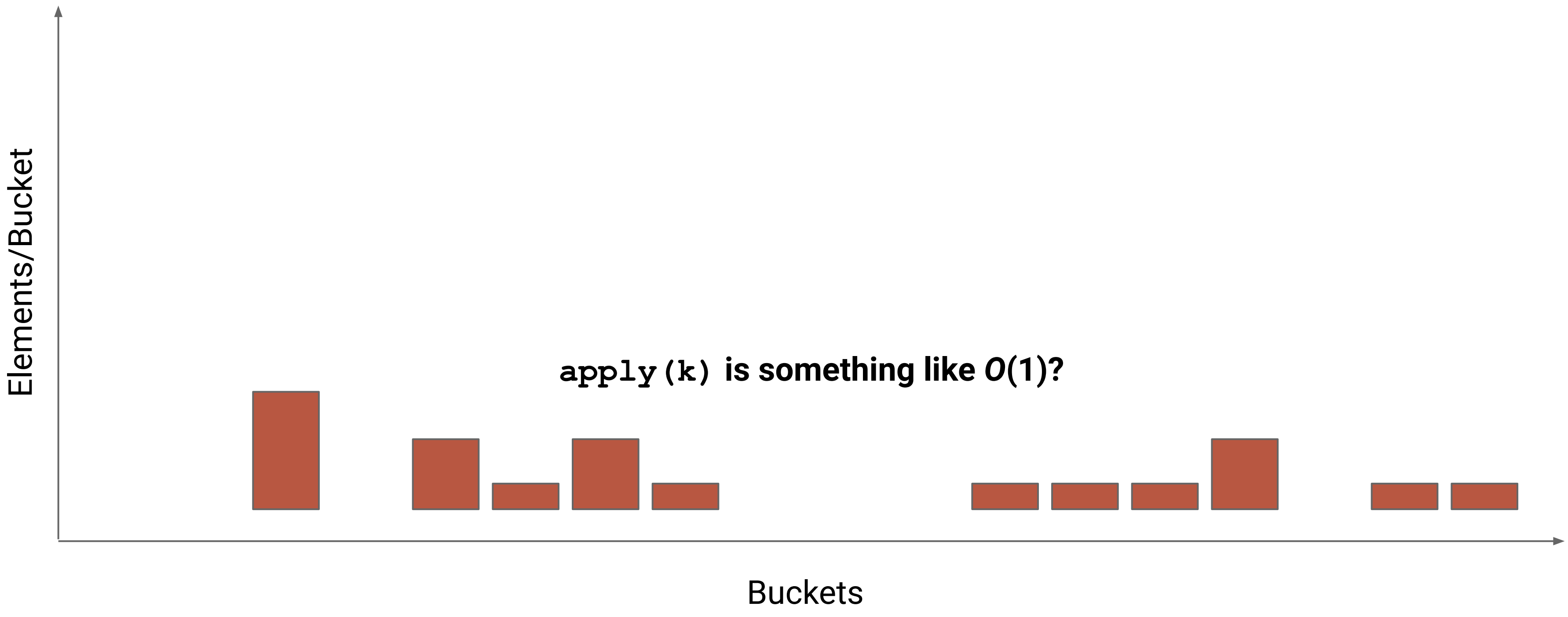
Picking a Hash Function



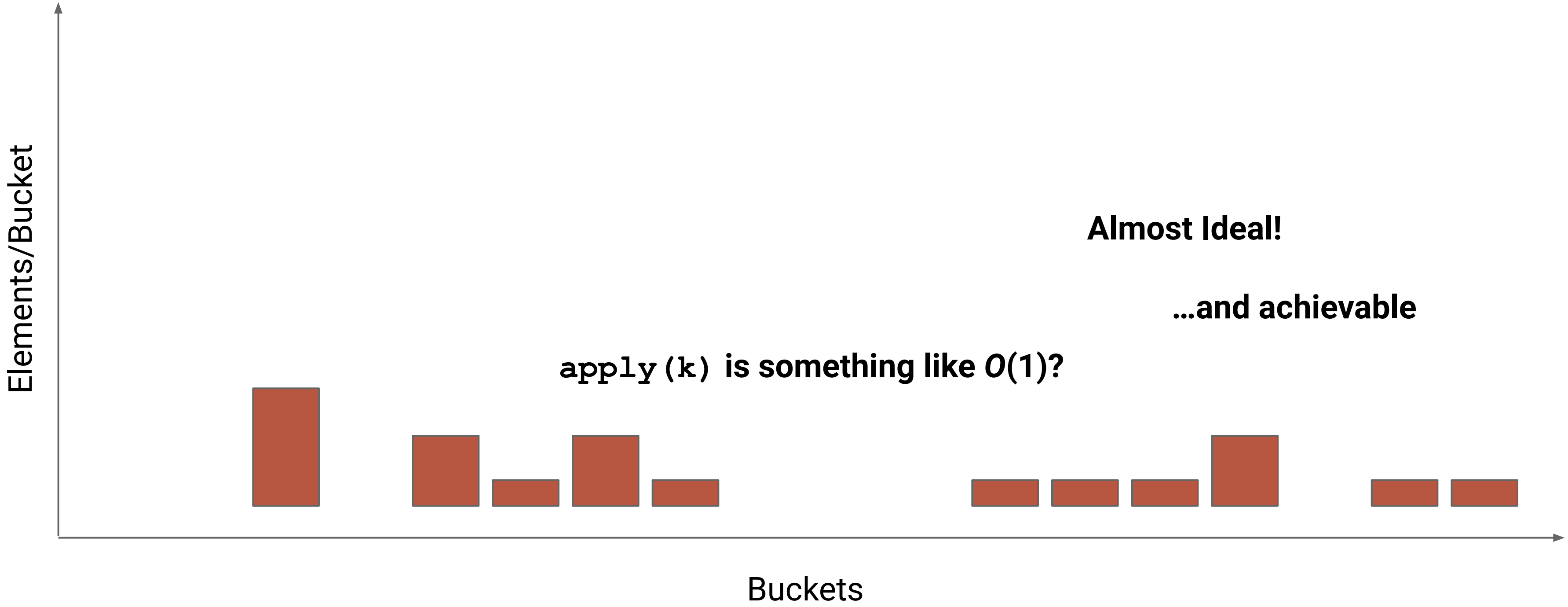
Picking a Hash Function



Picking a Hash Function



Picking a Hash Function

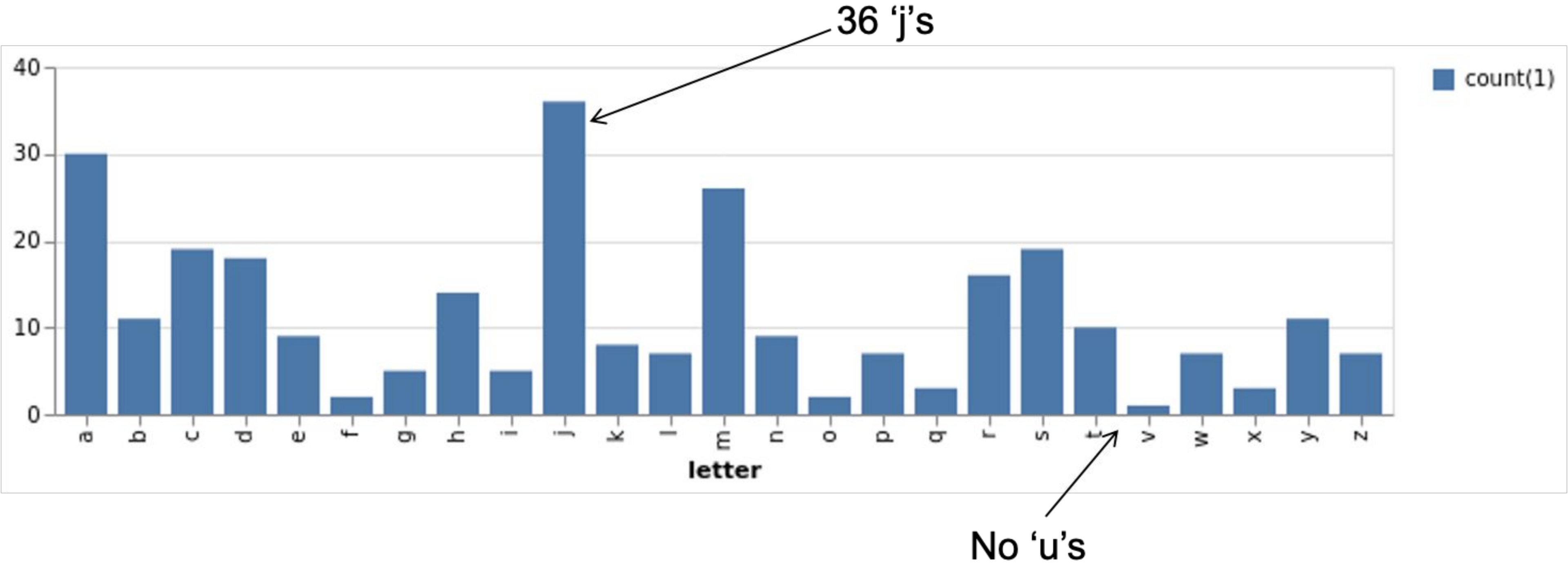


Other Functions

First Letter of UBIT Name

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

First Letter of UBIT Name



Other Functions

First Letter of UBIT Name

- Unevenly distributed, ***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 50m+ element array
- **Problem:** For reasonable ***N*** identity function returns something > ***N***

Other Functions

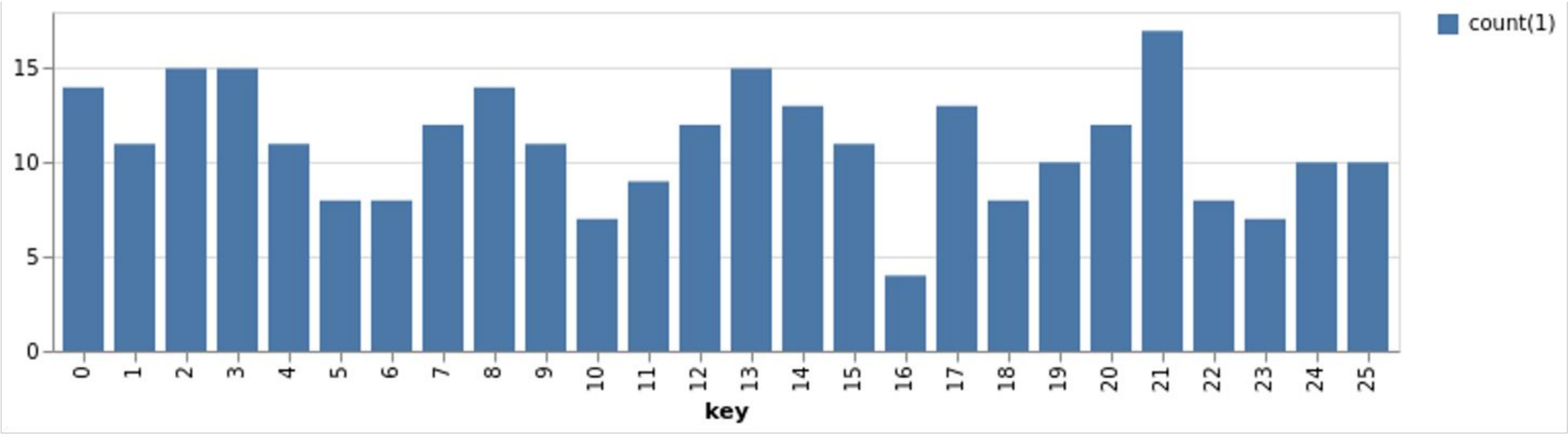
First Letter of UBIT Name

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

Identity Function on UBIT #

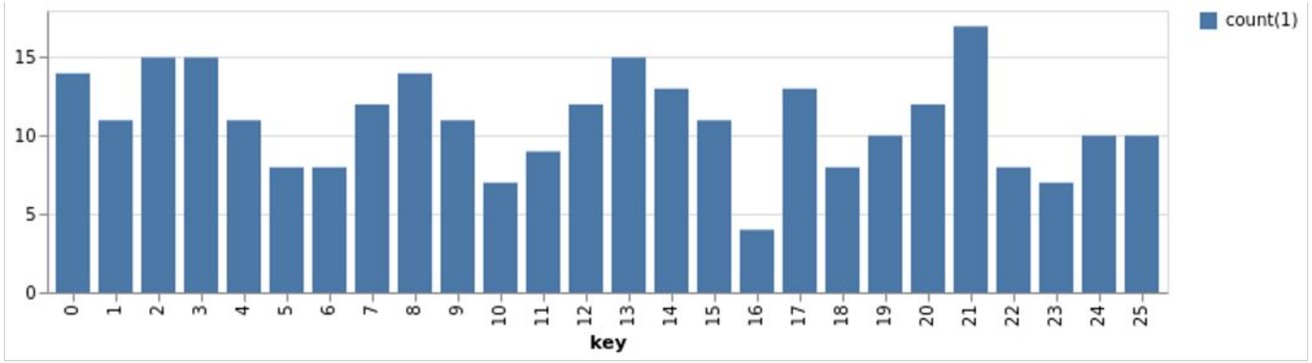
- Need a 50m+ element array
- **Problem:** For reasonable $\boldsymbol{N}$ identity function returns something > $\boldsymbol{N}$
- **Solution:** Cap return value of function to $\boldsymbol{N}$ with modulus
  - (x: Int) => x % N

Identity of UBIT # mod 26

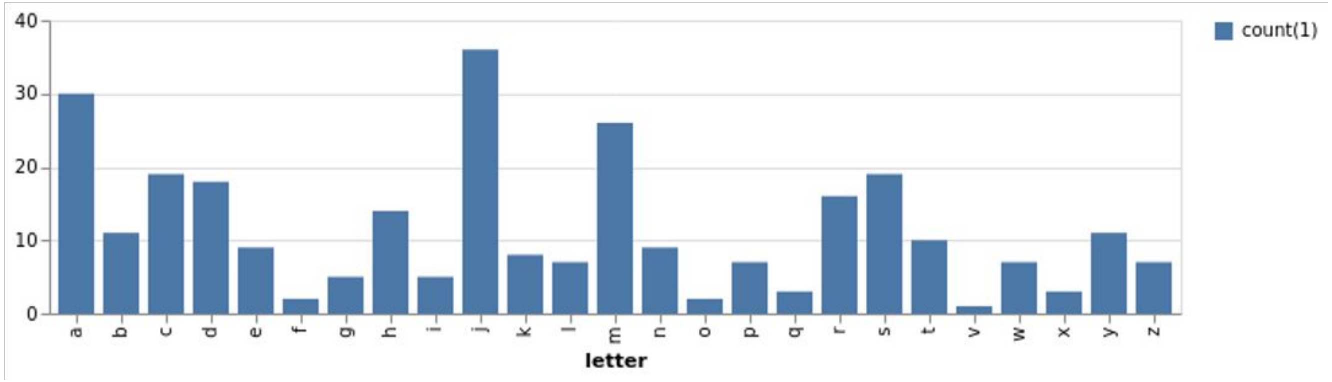


Comparison

UBIT # % 26



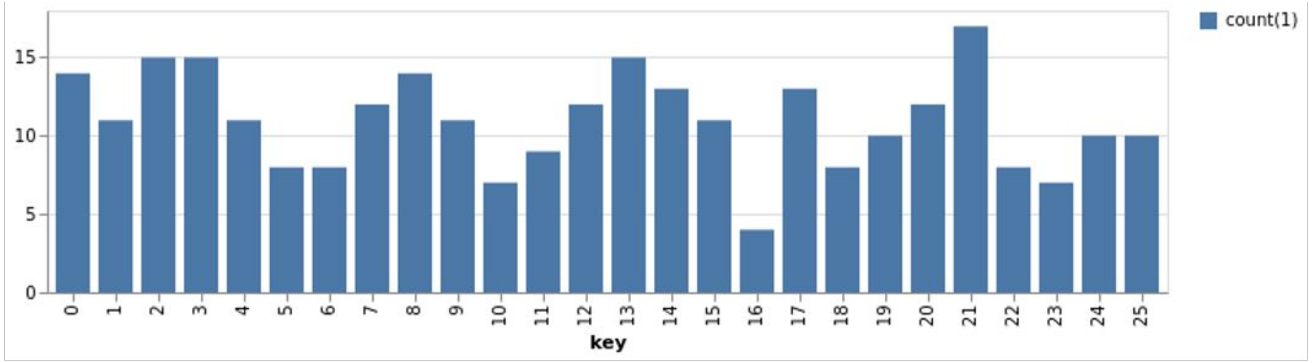
substr(UBITName, 0, 1)



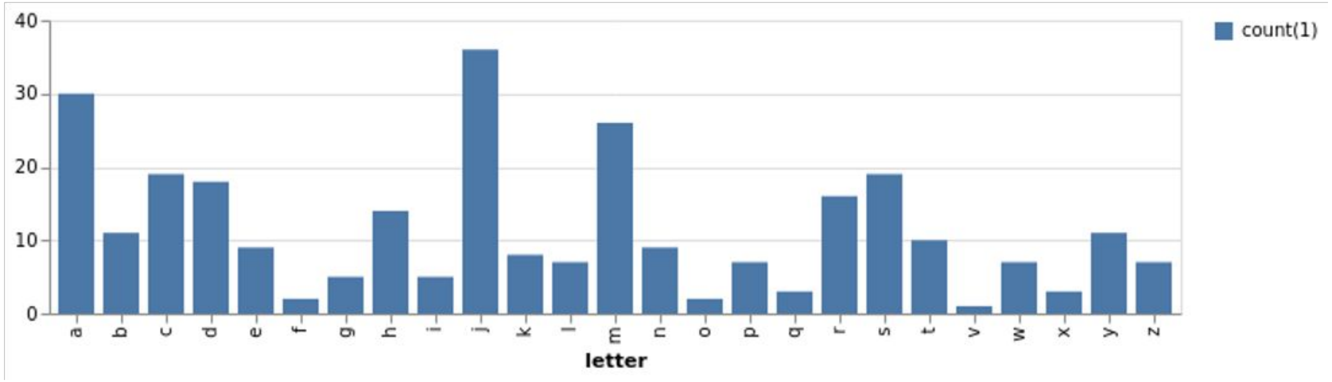
Comparison

UBIT # % 26

This still relies on UBIT # being "randomly distributed"



substr(UBITName, 0, 1)



Picking a Hash Function

Wacky Idea: Have $\boldsymbol{h(x)}$ return a random value in **[0,N)**

(This makes apply impossible…but bear with me)

Random Hash Function

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

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

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

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

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

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

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

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

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'))

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

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

Random Hash Function

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

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

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

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 uncorreleted 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?