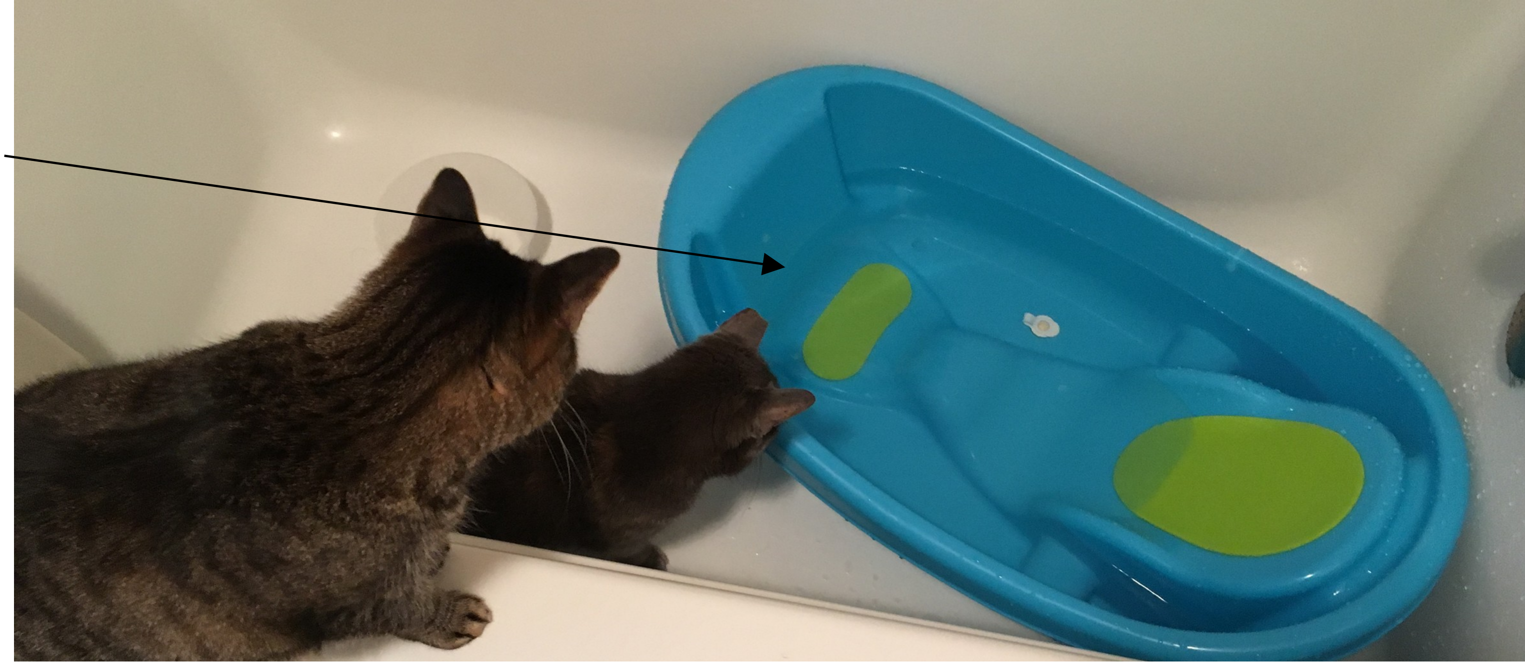
An unacceptable position



CSE 250
Lecture 29

Hash Tables

Maps

- Sets, Bags: Collections of Elements of type A
  - add(a: A)
  - remove(a: A)
  - apply(a: A)
    - set.apply(a:A): Boolean // is the element part of set
    - bag.apply(a:A): Int // # of copies of the element in bag
- Maps: Like Sets, but where A is a 2-tuple (key, value)
  - The identity of the element is determined by key

Maps

- Map[K, V]
  - add(k: K, v: V)   // also called put(k, v)
    - Insert (k, v) into the map
    - If an element associated with key k already exists, replace it.
  - remove(k): V
    - Remove the element associated with key k, return the corresponding value
  - apply(k: K): V   // also called get(k)
    - Return the value corresponding to key k.

Maps

- Map[K,V] as a Sorted Sequence
  - O(log(n)) apply (but very cache-friendly)
  - O(n) add
  - O(n) remove
- Map[K,V] as a balanced Binary Search Tree
  - O(log(n)) apply
  - O(log(n)) add
  - O(log(n)) remove
- Map[K,V] as a LSM tree

Finding Items

The most epensive part of finding records is **finding** them.

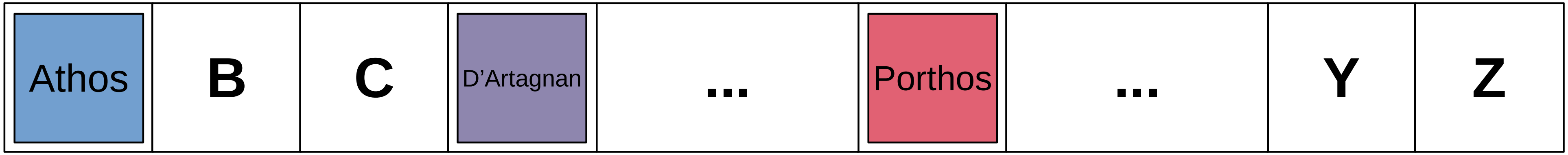
(i.e., where is the record located?)

So... skip the search

Alternative Idea: Assign Bins

- Create an array of size N
- Pick an O(1) function to assign each record a number in [0,N)
  - First letter of name → [0, 26)

Alternative Idea: Assign Bins



Alternative Idea: Assign ~~Bins~~ Buckets

- Pros
  - O(1) Insert
  - O(1) Find
  - O(1) Remove
- Cons
  - Wasted Space (Only 3/26 slots used)
  - Duplication (What about Aramis?)

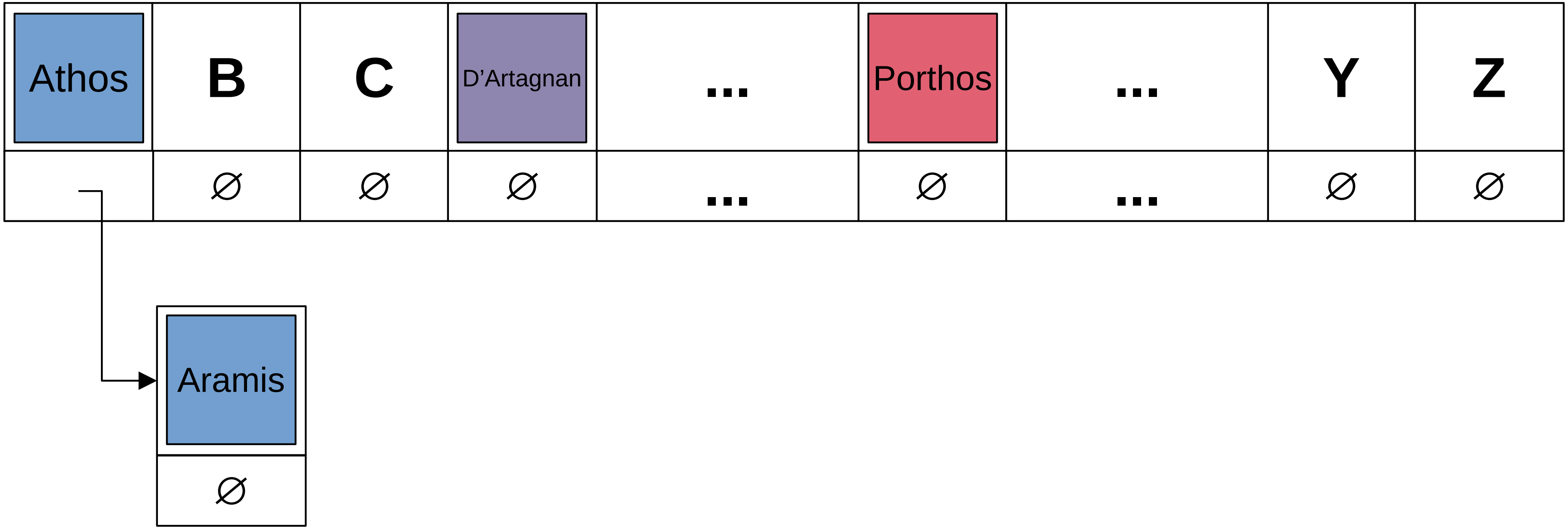
Bucket-Based Organization

- Wasted Space
  - Not ideal, but not wrong
  - O(1) access time might be worth it!
  - Also depends on choice of function (more on this later)
- Duplication
  - We need to deal with duplicates!

Duplication

- **Idea**: Make buckets bigger (e.g., B elements)
  - **Pro**: Up to B duplicates in a bucket, Still O(1) (O(B)) find
  - **Con**: No more than B duplicates in a bucket
- **Idea**: Make buckets arbitrarily large (e.g., Linked List)
  - **Pro**: No more overflow
  - **Con**: O(n) worst-case find.

Buckets + Linked Lists



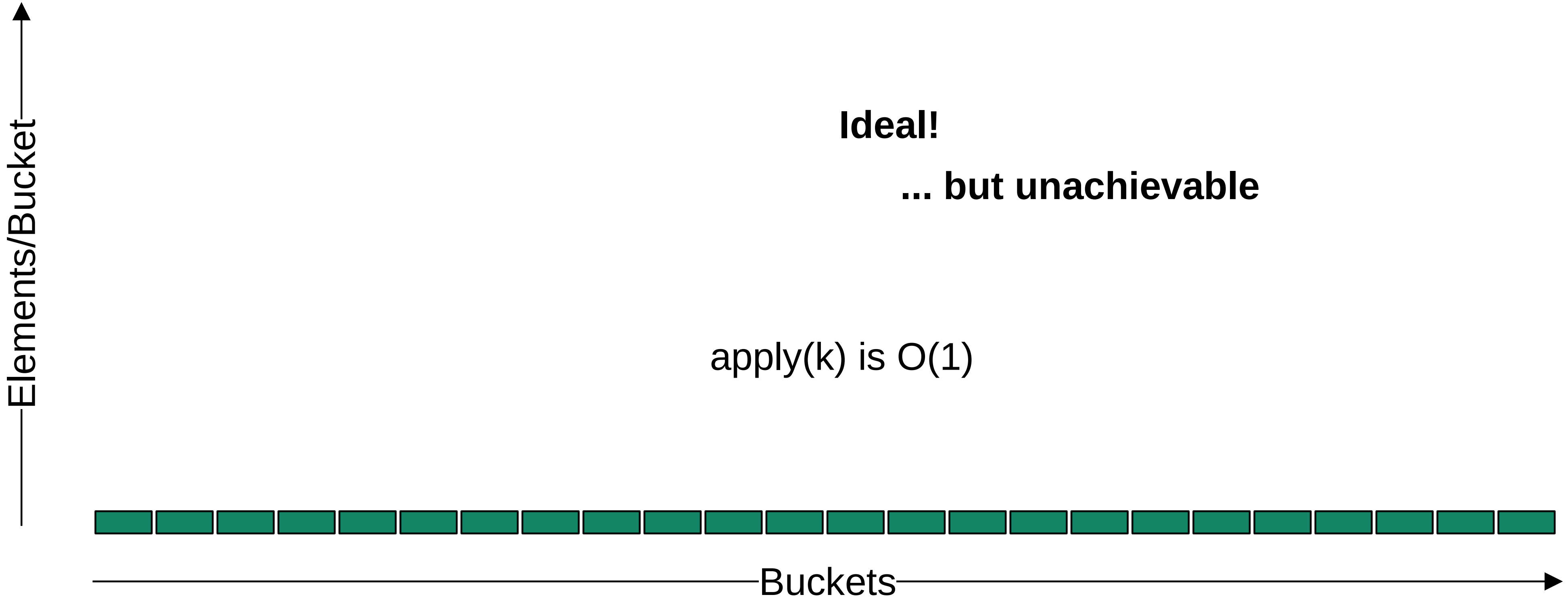
Buckets + Linked Lists

```
class BucketMap[K, V](_numBuckets: Int, _lookupFunction: K => Int) {

  val _buckets = new Array[ List[(K, V )] ](_numBuckets)
  for(i <- 1 until _numBuckets) { _buckets(i) = Nil }

  def apply(key: K): V = {
    val bucketPosition = _lookupFunction(k)
    var element = _buckets(bucketPosition)
    while(element != Nil){
      if(key == element.head._1) { return element.head._2; }
      element = element.tail
    }
    throw new NoSuchElementException(key.toString)
  }
}
```

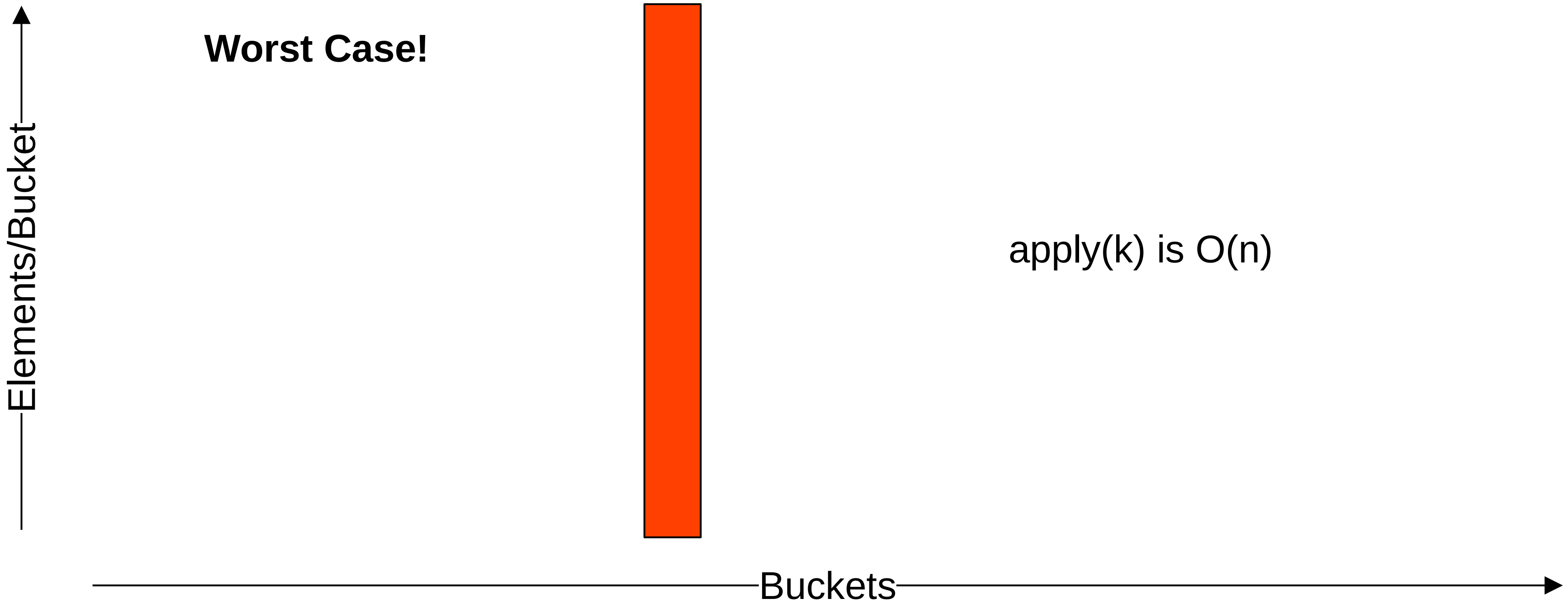
Picking a Lookup Function

- Desirable Features for h(x)
  - Fast
    - needs to be O(1)
  - "Unique"
    - As few duplicate bins as possible

Picking a Lookup Function

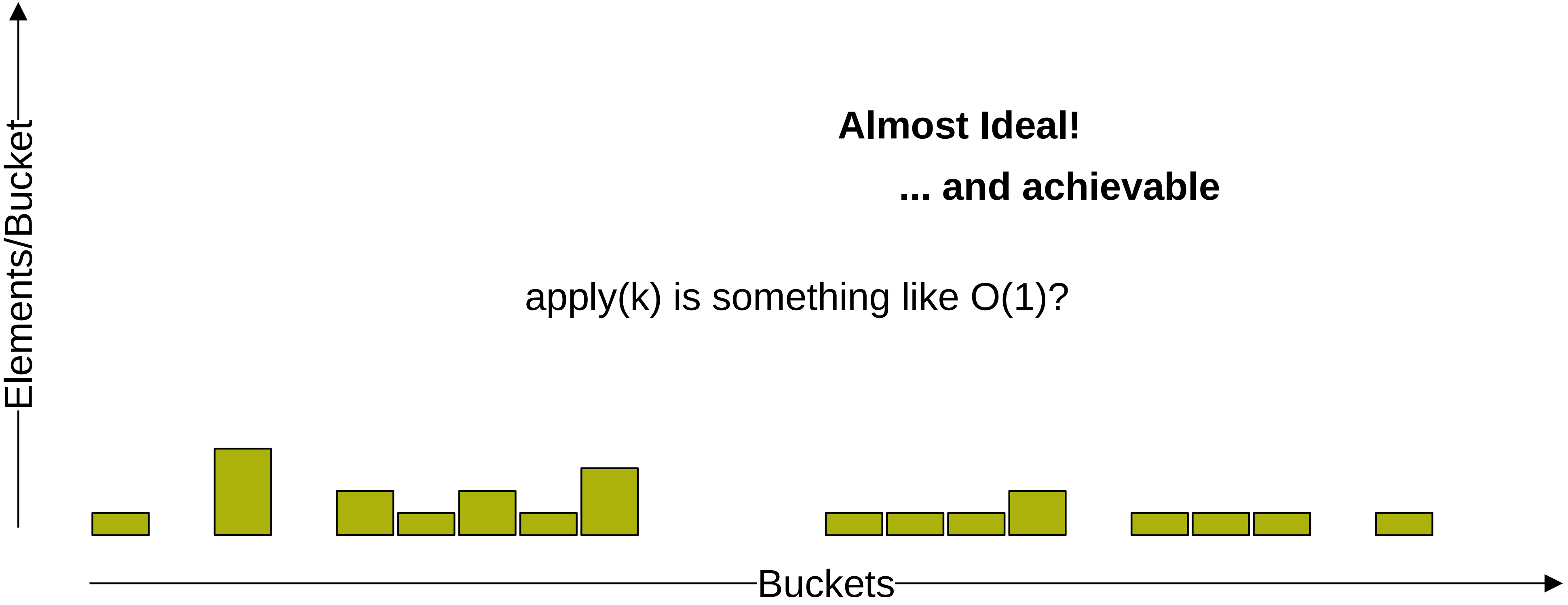


Picking a Lookup Function



apply(k) is O(n)

Picking a Lookup Function



Almost Ideal!

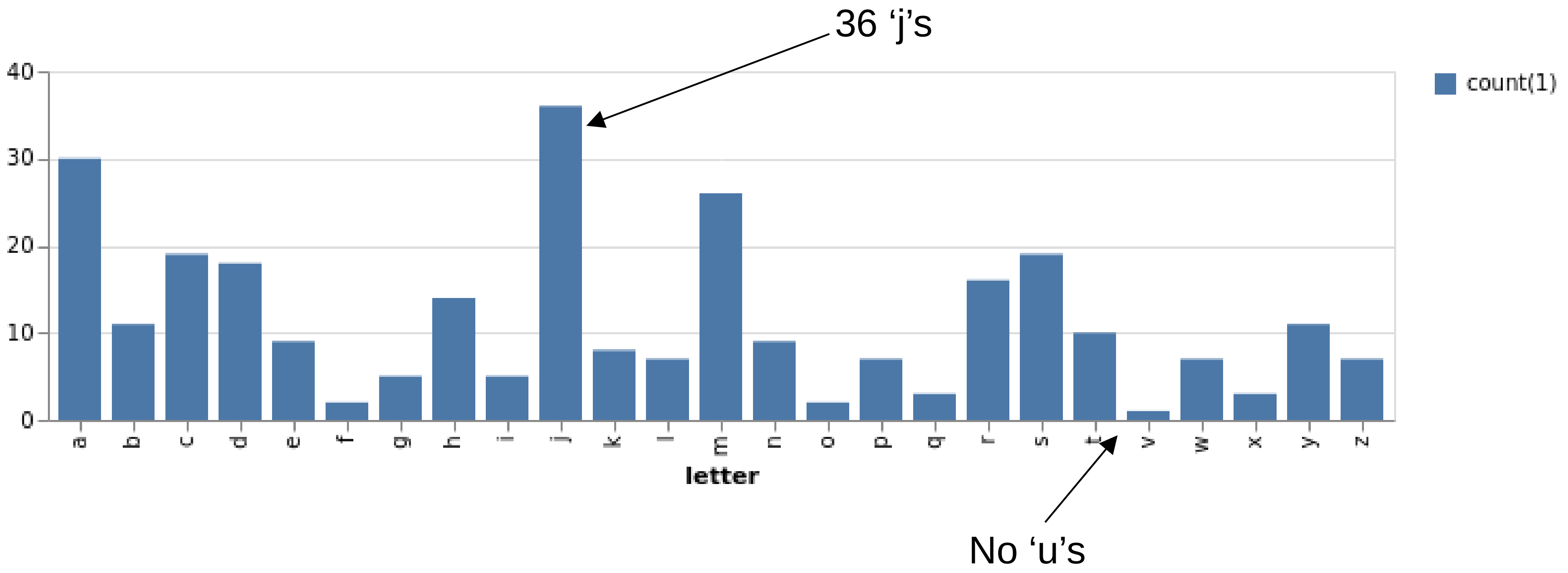
... and achievable

apply(k) is something like O(1)?

Other Functions

- First Letter of UBIT

First letter of UBIT



Other Functions

- First Letter of UBIT name
  - Unevenly Distributed: O(n) apply
- Identity Function on UBIT #
  - Need a 50m+ element array

Other Functions

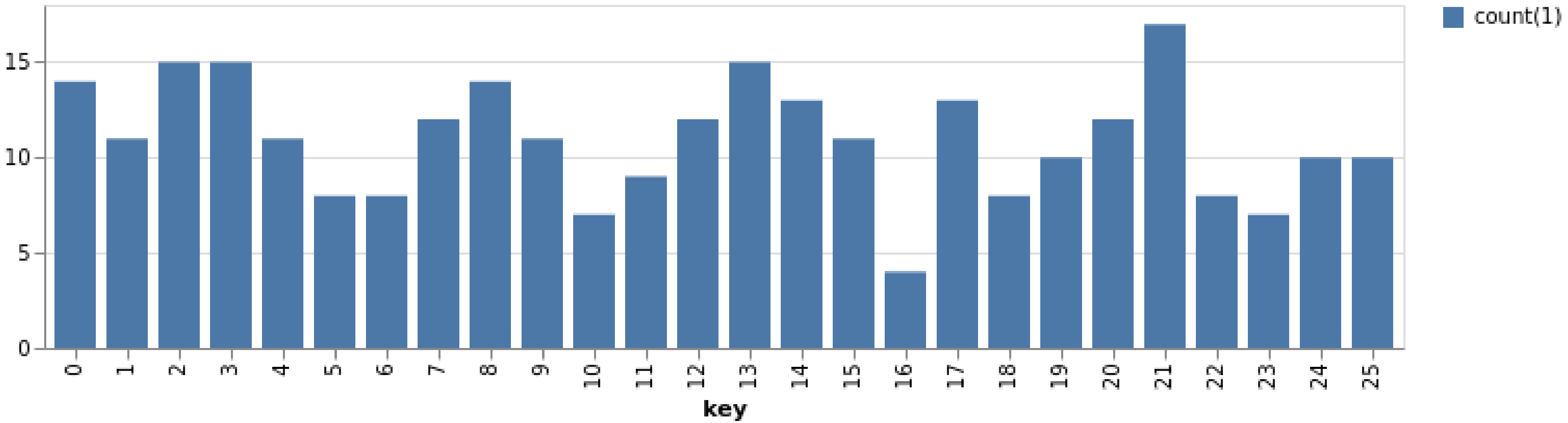
- Identity Function: `(x: Int) => x`
  - **Problem**: Can return values over N
  - **Solution**: Cap return value by Modulus with N
    - `(x: Int) => x % N`

Modulus

0	1	2	3	4	5	6

0	1	2	3	4	5	6
7	8	9	10	11	12	13
14	15	16	17	18	19	20

Modulus

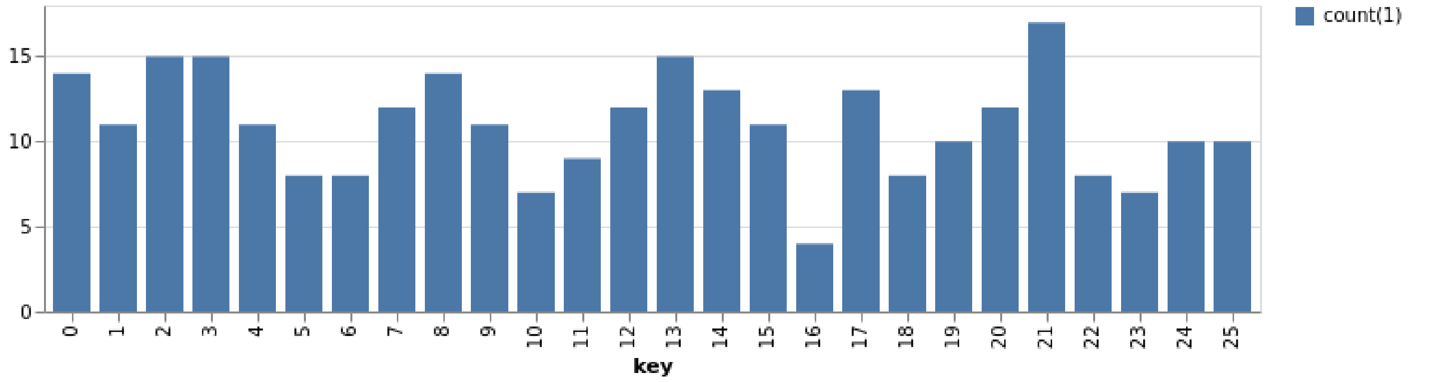


count(1)
0
5
10
15
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
key

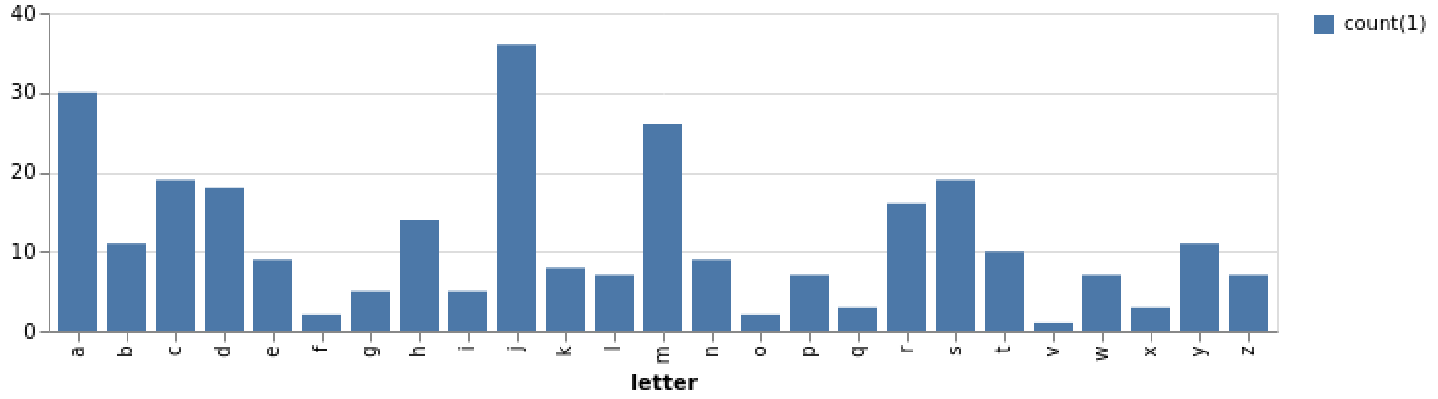
Modulus

But still relies on UBIT # being random!

UBIT # % 26



substr(UBITName, 0, 1)



Picking a Lookup Function

- **Wacky Idea**: Have h(x) return a random value in [0, N)
  - Random.nextInt % N

    (Yes, it makes apply impossible, but bear with me)

Picking a Lookup Function

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

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

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

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

Picking a Lookup 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'

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

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

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

Picking a Lookup 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(): $O\left(\frac{n}{N}\right)$

Worst-Case Runtime of insert, apply, remove(): $O\left(n\right)$

Hash Functions

- Examples
  - SHA256 ← used by GIT
  - MD5, BCRYPT ← used by unix login, apt
  - MurmurHash3 ← used by Scala
- hash(x) is pseudorandom
  1) hash(x) ~ uniform random value in [0, INT_MAX)
  2) hash(x) always returns the same value
  3) hash(x) uncorrelated with hash(y) for x ≠ y

Using Hash Functions

- hash(x: Int): Int
  - What about strings?

Arbitrary starting constant
(hash(“”))

```
def hashString(str: String): Int = {
  var accumulator: Int = SEED
  for(character <- str){
    accumulator = hash(accumulator * character.toInt)
  }
  return accumulator
}
```

call hash() str.length times

(simplified, don’t actually do exactly this)

Hash Functions

- hash(x: Object): Int
  - In Java/Scala, call x.hashCode

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 happens when this constraint is violated?