CSE 250: Sets, Hash Tables

Lecture 30

Nov 13, 2023

Reminders

- PA3 to be released Wednesday
  - "Join" two datasets together efficiently.
  - De-anonymize "public" data

The Set ADT

A collection of unique elements (of type `E`)

- `public boolean add(E a)`
  Add an element `a` to the set and return true. Do nothing and return false if it is already present.
- `public boolean remove(E a)`
  Remove an element `a` from the set and return true. Do nothing and return false if the element is not in the set.
- `public boolean contains(E a)`
  Return true if and only if the element `a` is part of the set.
- `public int size()`
  Return the number of elements in the set.

How do we implement a set?

- List (Array or Linked)?

ListSet

- `public boolean add(E a)`
  Append the element to the list. $O(1)$ (amortized if ArrayList)
- `public boolean remove(E a)`
  Find the element and remove it. $O(N)$
- `public boolean contains(E a)`
  Find the element. $O(N)$
- `public int size()`
  Return the size of the list. $O(1)$

How do we implement a set?

- ~~List (Array or Linked)?~~
- Sorted ArrayList?

SortedListSet

- `public boolean add(E a)`
  Insert the element into the list. $O(N)$
- `public boolean remove(E a)`
  Find the element and remove it. $O(N)$
- `public boolean contains(E a)`
  Find the element. $O(\log N)$
- `public int size()`
  Return the size of the list. $O(1)$

How do we implement a set?

- ~~List (Array or Linked)?~~
- ~~Sorted ArrayList?~~
- Balanced Binary Search Tree (AVL, Red-Black)

TreeSet

- `public boolean add(E a)`
  Insert the element into the tree. $O(\log N)$
- `public boolean remove(E a)`
  Find the element in the tree and remove it. $O(\log N)$
- `public boolean contains(E a)`
  Find the element in the tree. $O(\log N)$
- `public int size()`
  Return a pre-computed size. $O(1)$

How do we implement a set?

- ~~List (Array or Linked)?~~
- ~~Sorted ArrayList?~~
- Balanced Binary Search Tree (AVL, Red-Black) $O(\log N)$
- ???

Finding Items

The most expensive part of finding an element is finding it.

- Searching through every element of a List
- Binary Search on an ArrayList
- Searching through a (balanced) BST.

... so let's skip the search.

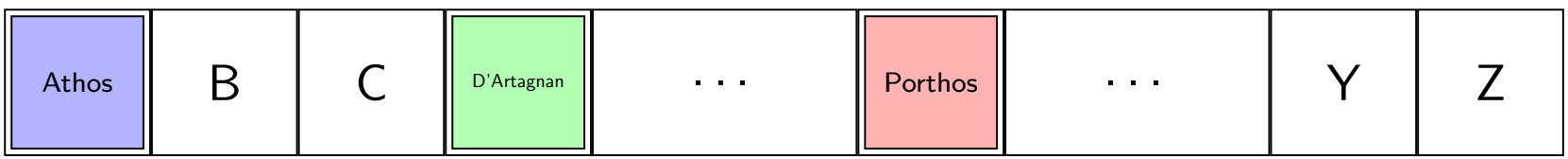
Bucketing Elements

- Create an array `data` of size N
- Pick a function $\texttt{h}(e)$ that assigns each element e to a number in $[0, N)$
  - We want the function $\texttt{h}(e)$ to be $O(1)$
- Only look for the element e at $\texttt{data}[\texttt{h}(e)]$.

What are some good functions $h(e)$?

- First letter of string? ($N = 26$)

Bucketing Elements



Bucketing Elements

The Good

- $O(1)$ Insert
- $O(1)$ Find
- $O(1)$ Remove

The Bad

- Wasted Space (Only 3/26 slots used)
- Duplication (What about Aramis?)

Bucketing Elements

Wasting Space

- Note ideal, but not wrong.
  - $O(1)$ might be worth a little wasted space.
- ... also depends on $\texttt{h}(e)$

Duplication

- ... is a real problem.

Duplication

Idea: Make the buckets bigger (e.g., array of B elements)

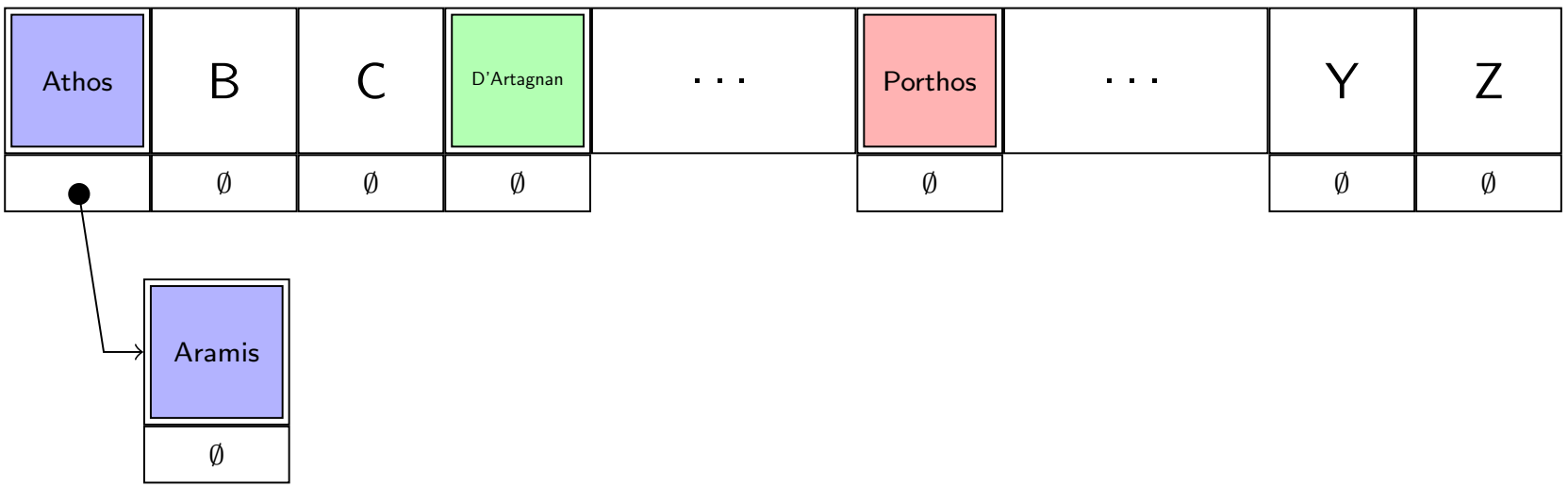
- **Pro:** Up to B duplicates in a bucket
  $O(B) = O(1)$ find
- **Con:** What happens if we need $> B$ elements in a bucket?

Duplication

Idea: A linked list of buckets

- **Pro:** No more overflow
- **Con:** `find` becomes (unqualified) $O(n)$

Bucketing Elements with Linked Lists



Bucketing Elements with Linked Lists

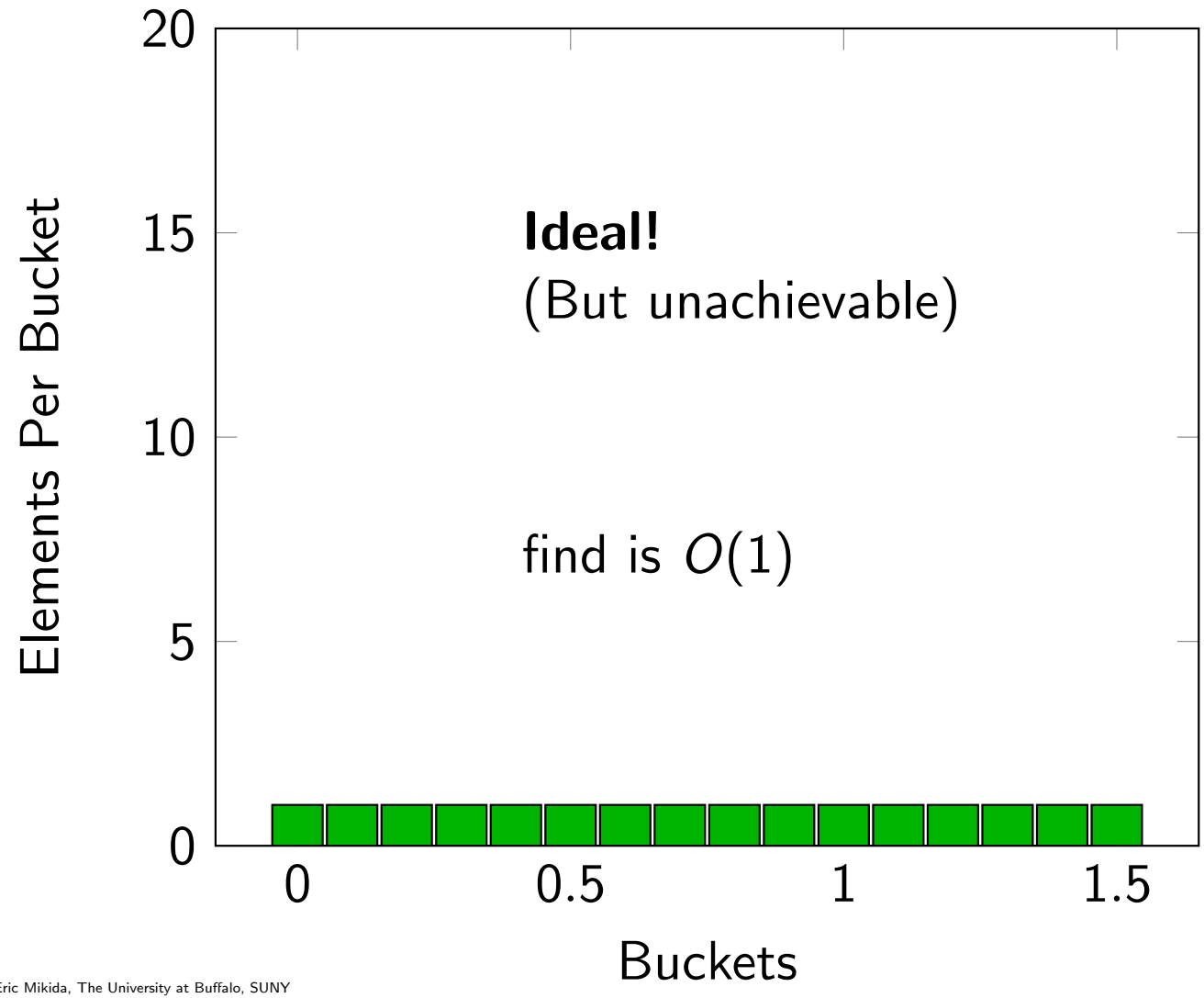
```
1     public class LinkedListNode<E>
2     {
3       public E value;
4       public Optional<LinkedListNode<E>> next = Optional.empty();
5     }
6
7     public class LinkedListLookupTable<E>
8     {
9       public Optional<LinkedListNode>[] elements;
10    }
```

Bucketing Elements with Linked Lists

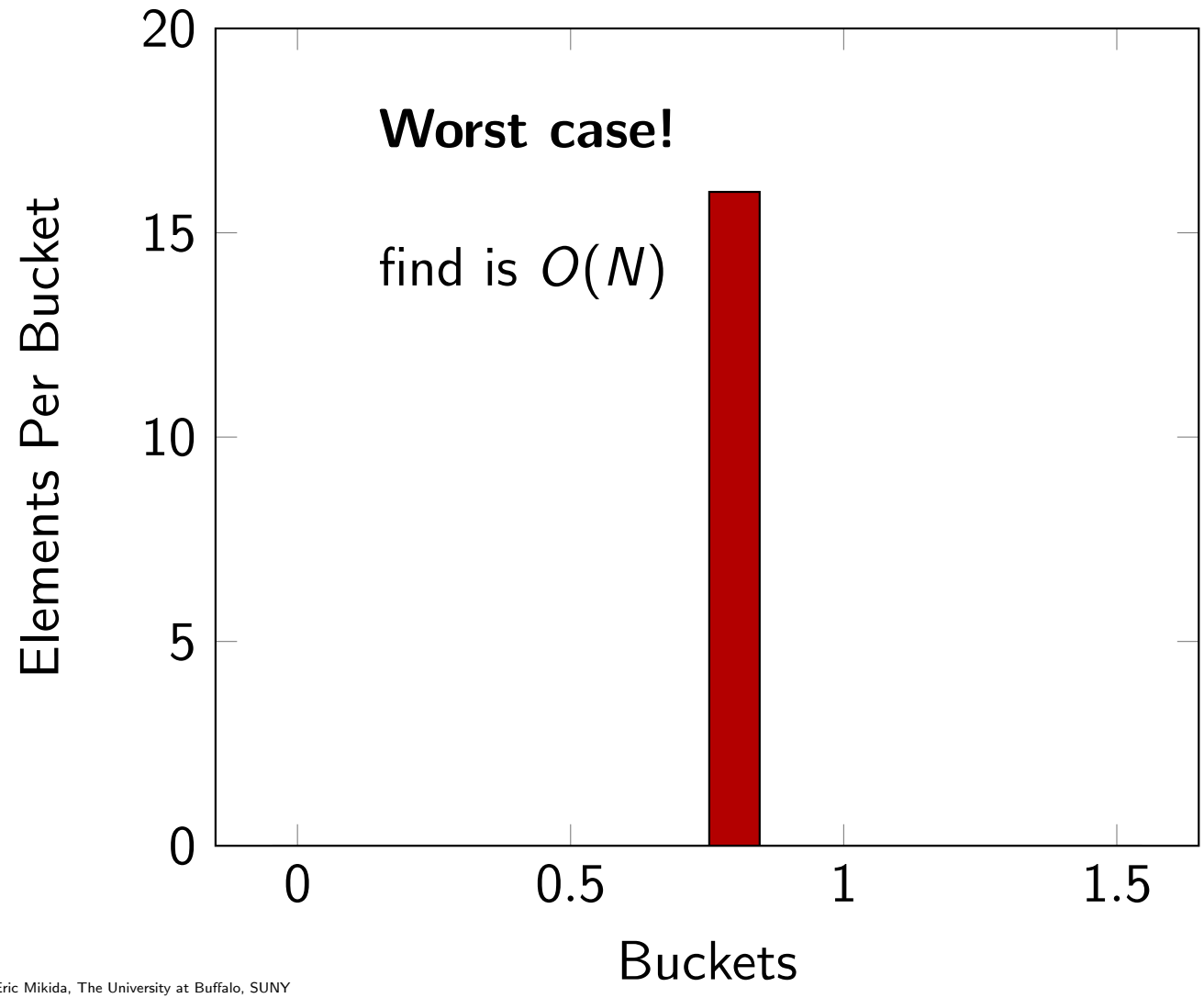
Find

- Find the linked list at position $\texttt{h}(e)$
- Find e in the linked list

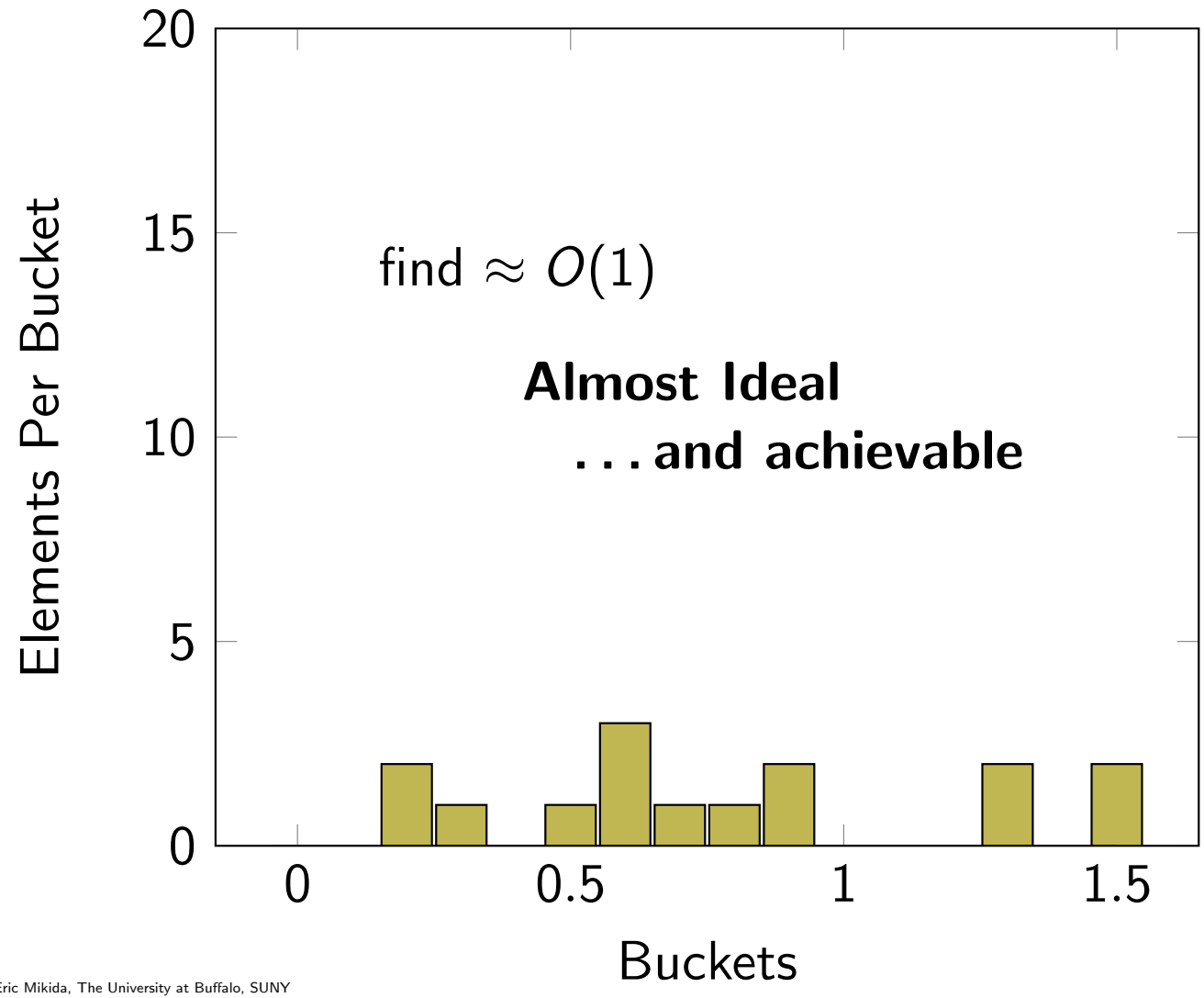
Picking a lookup function



Picking a lookup function



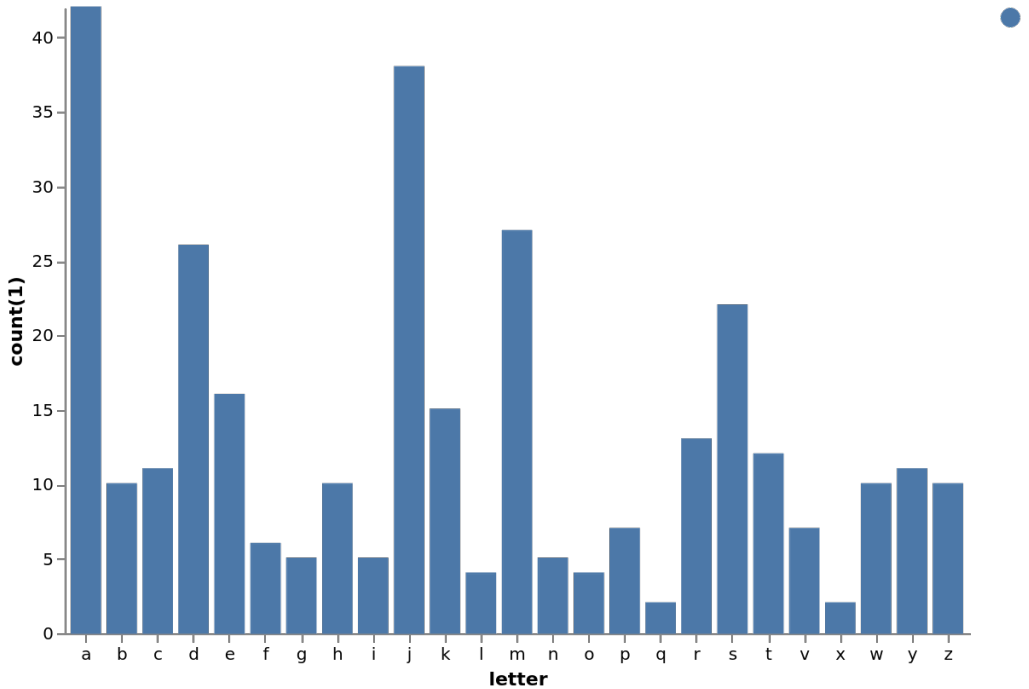
Picking a lookup function



Bucketing You

- First letter (of your email address)

Bucketing You

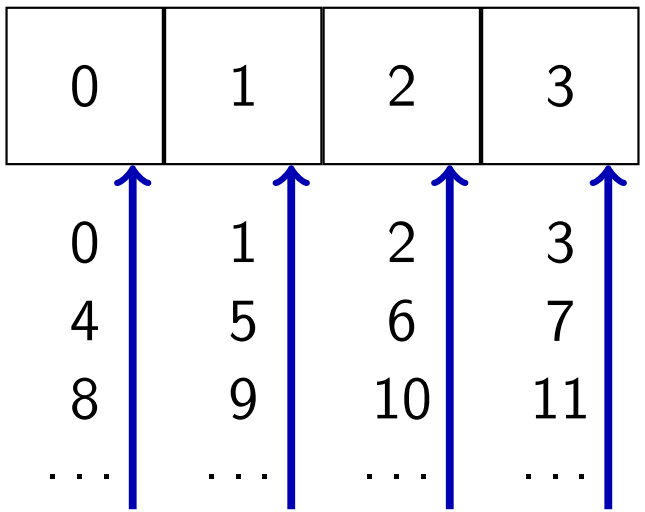


- 42 'a' names
- No 'u' names

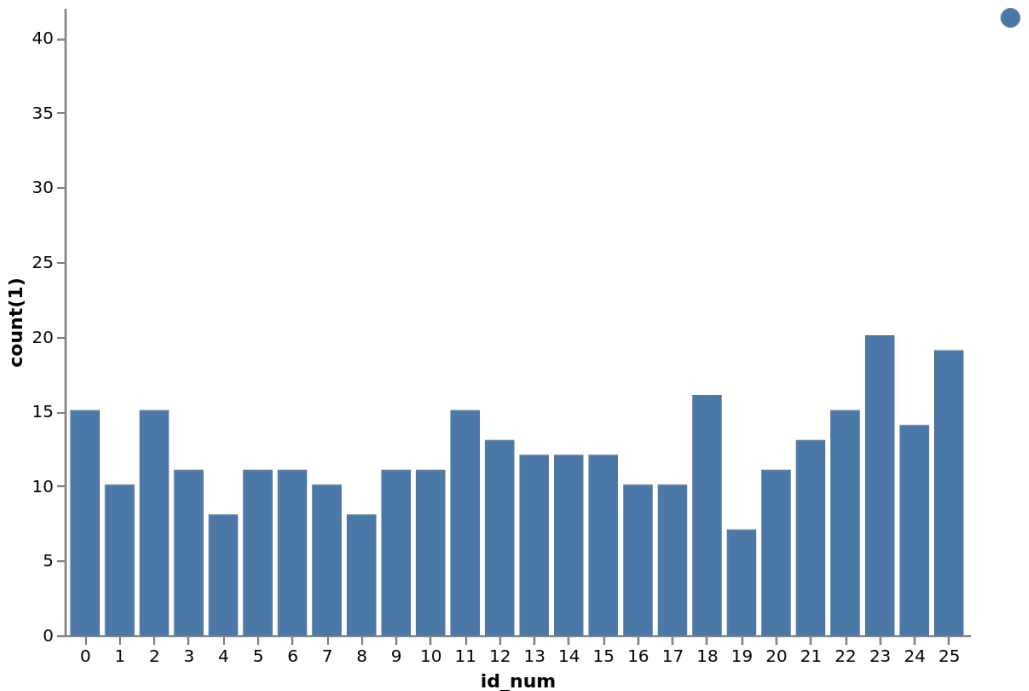
Bucketing You

- ~~First letter (of your email address)~~
  - Unevenly distributed: $\approx O(N)$ apply
- UB ID # ('person number')
  - Need a 50m+ element array

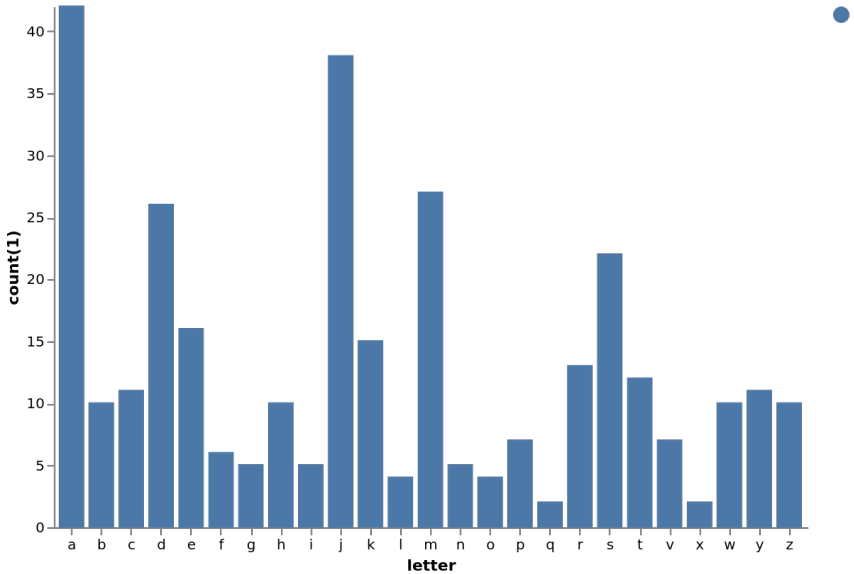
Modulus



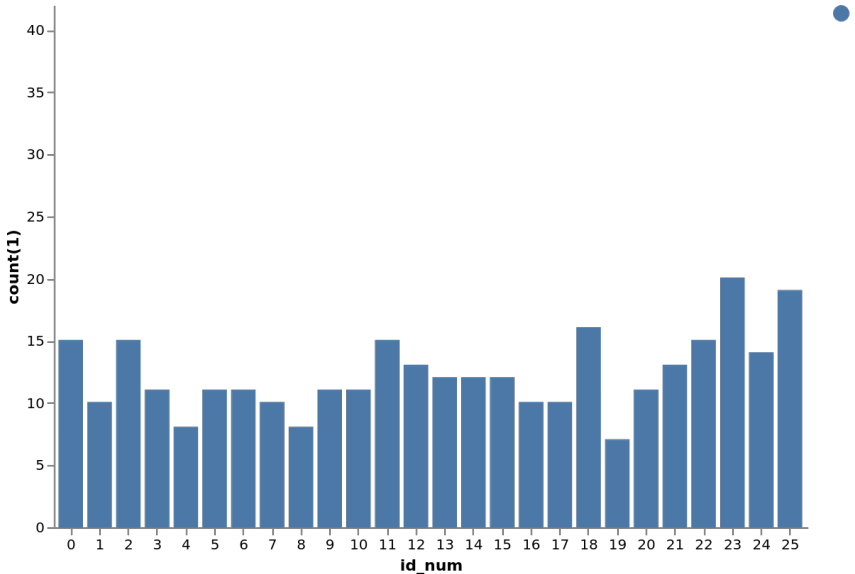
Modulus



Bucketing You



substr(email, 0, 1)



mod(id, 26)

Why is mod(id) good?

`mod(id, N)` is basically random.

Idea: What if $\texttt{h}(x)$ returns a random value.
(Yes, makes finding records impossible, but bear with me)

Picking a lookup function

- $N =$ number of elements in any bucket
- $B =$ number of buckets
- $b_{i,j} = \begin{cases} 1 & \textbf{if element } i \text{ is assigned to bucket } j \\ 0 & \textbf{otherwise} \end{cases}$

$$\mathbb{E}[b_{i,j}] = \frac{1}{B}$$

Picking a lookup function

- $N =$ number of elements in any bucket
- $B =$ number of buckets
- $b_{i,j} = \begin{cases} 1 & \textbf{if element } i \textbf{ is assigned to bucket } j \\ 0 & \textbf{otherwise} \end{cases}$

The Expected Number of Elements in Bucket j

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

... if $b_{i,j}$ and $b_{i',j}$ are uncorrelated for $i \neq i'$

($\mathtt{h}(i)$ can't be related to $\mathtt{h}(i')$)

Picking a lookup function

- $N =$ number of elements in any bucket
- $B =$ number of buckets
- $b_{i,j} = \begin{cases} 1 & \textbf{if element } i \textbf{ is assigned to bucket } j \\ 0 & \textbf{otherwise} \end{cases}$

- **Expected** Runtime of `insert`, `find`, `remove`: $O\left(\frac{N}{B}\right)$
- **Unqualified** Runtime of `insert`, `find`, `remove`: $O(N)$

... if we have a random-ish $\texttt{h}(e)$

Hash Functions

Example Hash Functions

- **SHA256** (used by GIT)
- **MD5**, **BCrypt** (used by unix login, apt)
- **MurmurHash3** (used by Scala)

`hash(e)` is pseudorandom

1. `hash(e)` $\sim$ uniform random value in $[0, \texttt{Integer.MAX\_VALUE})$
2. `hash(e)` always returns the same value for the same `e`
3. `hash(e)` is uncorrelated with `hash(e')` for `e` $\neq$ `e'`

Using Hash Functions

Basic Hash: `public int hash(int e)`

- Integers: `hash`(e) mod B gets the bucket of e
- Strings: ???

```
1  public int hashString(String str)
2  {
3    int accumulator = SEED;
4    for(c : str.toCharArray())
5    {
6      accumulator = hash(accumulator + c)
7    }
8    return accumulator
9  }
```

(simplified... don't actually do this)

Using Hash Functions in Java

For any object `x`, call `x.hashCode`

HashSet

- `public boolean add(E a)`
  Insert the element into the list at `hash`$(a) \mod B$.
  Expected $O\left(\frac{N}{B}\right)$
- `public boolean remove(E a)`
  Find the element in the list at `hash`$(a) \mod B$ and remove it.
  Expected $O\left(\frac{N}{B}\right)$
- `public boolean contains(E a)`
  Find the element in the list at `hash`$(a) \mod B$.
  Expected $O\left(\frac{N}{B}\right)$
- `public int size()`
  Return a pre-computed size. $O(1)$

Load Factor

Most operations are $O\left(\frac{N}{B}\right)$, let's call $\alpha = \frac{N}{B}$ the load factor.

Idea: Make α a constant

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

How do we repair the data structure when the constraint is violated?