Hashing

Apply to Section Lead!

http://cs198.stanford.edu

YEAH Hours

- YEAH Hours for Priority Queue are tomorrow from 4:15 – 5:45PM, 380-380C.
- Learn more about priority queues and linked lists!
- Get pointers about the trickier parts of the assignment.

The Story So Far

- We have now seen two approaches to implementing collections classes:
  - Dynamic arrays: allocating space and doubling it as needed.
  - Linked lists: Allocating small chunks of space one at a time.
- These approaches are good for **linear structures**, where the elements are stored in some order.

Associative Structures

- Not all structures are linear.
- How do we implement **`Map`**, **`Set`**, and **`Lexicon`**?
- There are many options, as you'll see in the next two weeks:
  - Hash tables.
  - Binary search trees.
  - Tries.
  - DAWGs.
- Today we will focus on implementing **`Map`**.

An Initial Implementation

- One simple implementation of **`Map`** would be to store an array of key/value pairs.
- To look up the value associated with a key, scan across the array and see if it is present.
- To insert a key/value pair, check if the key is mapped. If so, update it. If not, add a new key/value pair.

Kitty	Puppy	Ibex	Dikdik
Awww...	Cute!	Huggable	Yay!

An Initial Implementation

- One simple implementation of **`Map`** would be to store an array of key/value pairs.
- To look up the value associated with a key, scan across the array and see if it is present.
- To insert a key/value pair, check if the key is mapped. If so, update it. If not, add a new key/value pair.

Kitty	Puppy	Ibex	Dikdik	Hagfish
Awww...	Cute!	Huggable	Yay!	Ewww..

An Initial Implementation

- One simple implementation of **`Map`** would be to store an array of key/value pairs.
- To look up the value associated with a key, scan across the array and see if it is present.
- To insert a key/value pair, check if the key is mapped. If so, update it. If not, add a new key/value pair.

Kitty	Puppy	Ibex	Dikdik	Hagfish
Awww...	Really Cute!	Huggable	Yay!	Ewww..

Analyzing this Approach

- What is the big-O time complexity of inserting a value?
- Answer: **O(*n*)**.
- What is the big-O time complexity of looking up a value?
- Answer: **O(*n*)**.

Knowing Where to Look

- Our linked-list **`Stack`** implementation has O(1) push, pop, and top.
- Why is this?
- Know exactly where to look to find or insert a value.
- **`Queue`** implementation was O(n) for enqueue, but was improved to O(1) by adding extra information about where to insert.

Knowing Where to Look

- Our **`Vector`** supports O(1) lookups anywhere, even if there are *n* elements.
- Why is this?
- Know exactly where to look to find it.
- It's at position *n* in the array.

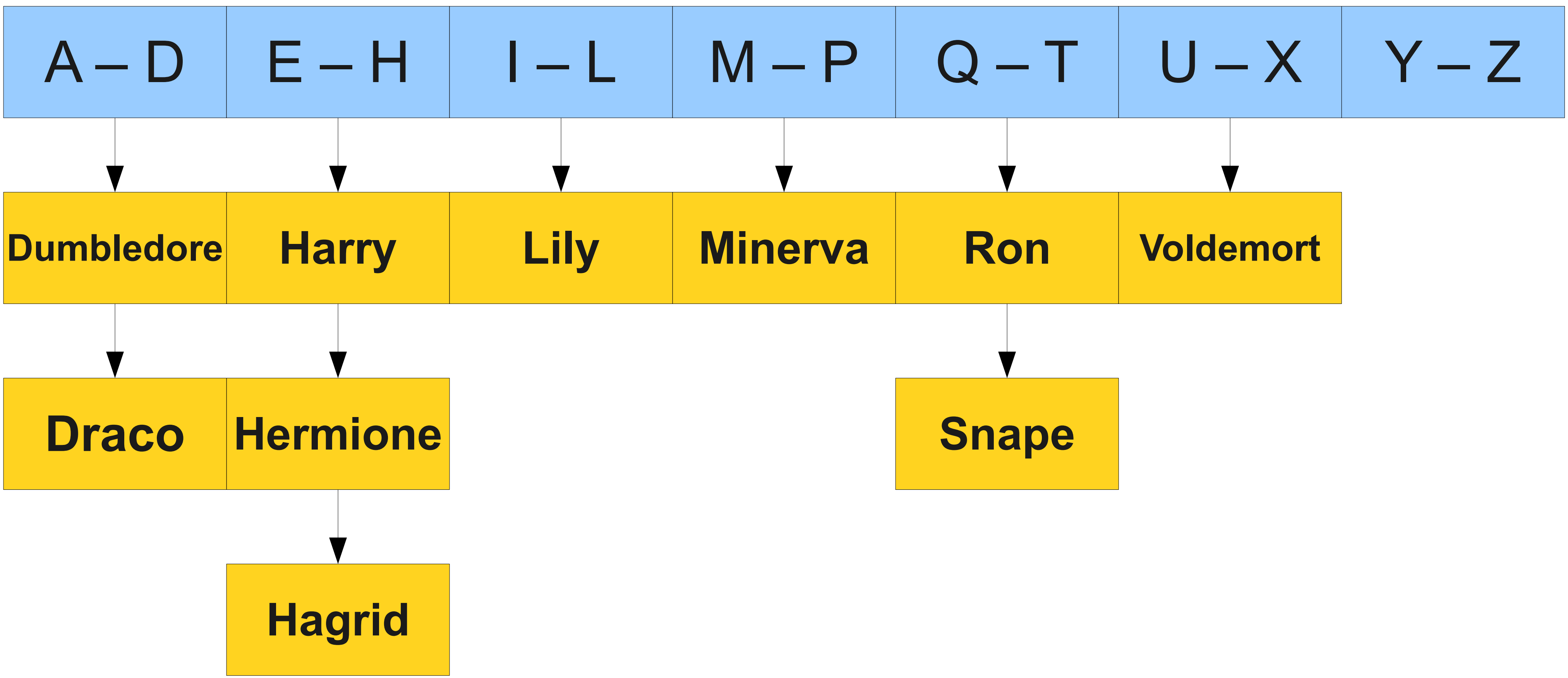
An Example: Clothes

For Large Values of *n*

Overview of Our Approach

- To store key/value pairs efficiently, we will do the following:
  - Create a lot of **buckets** into which key/value pairs can be distributed.
  - Choose a rule for assigning specific keys into specific buckets.
  - To look up the value associated with a key:
    - Jump into the bucket containing that key.
    - Look at all the values in the bucket until you find the one associated with the key.

Overview of Our Approach

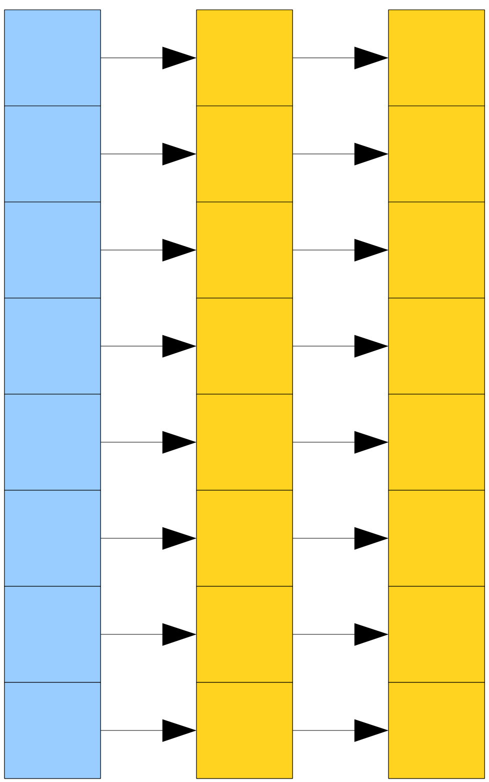


Hashing

- The rule we use to associate keys (in our case, strings) with specific buckets is called a **hash function**.
- Data structures that distribute items using a hash function are called **hash tables**.

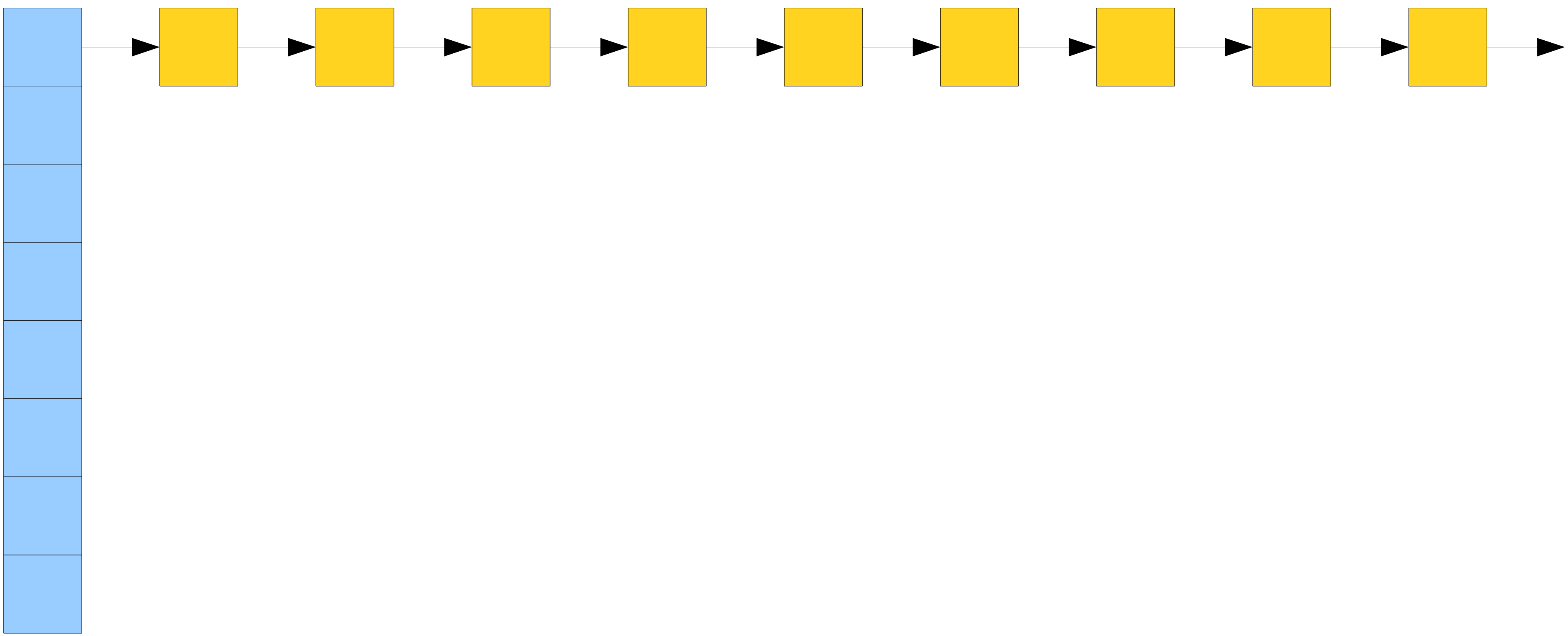
Distributing Keys

- When distributing keys into buckets, we want the distribution to be as random as possible.
- Best-case: totally even spread.
- Worst-case: everything bunched up.



Distributing Keys

- When distributing keys into buckets, we want the distribution to be as random as possible.
- Best-case: totally even spread.
- Worst-case: everything bunched up.



Distributing Keys

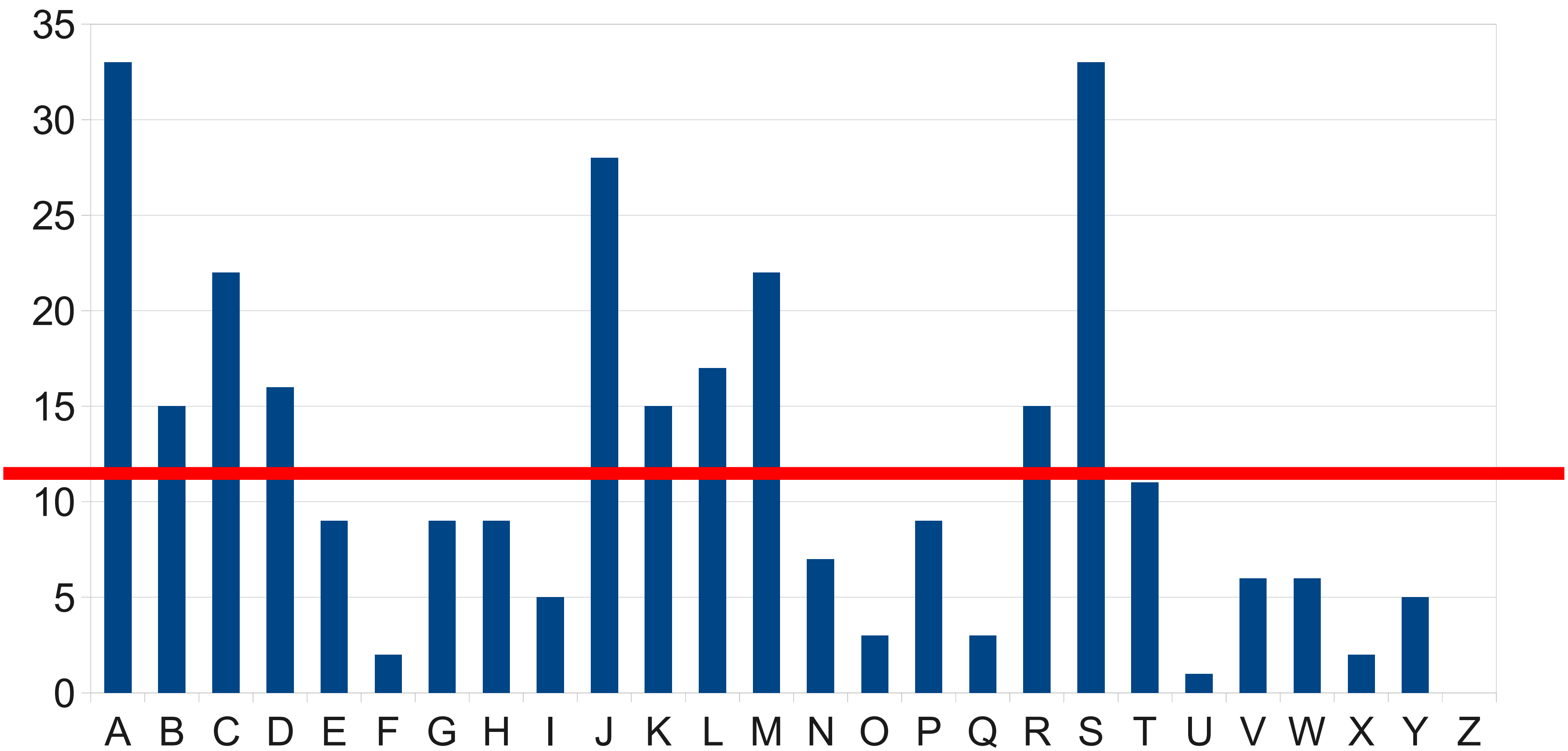
- We want to choose a function that will distribute elements as randomly as possible to try to guarantee a nice, even spread.
- We can't actually distribute them randomly.
  - Why not?
- Instead, we need a function that will really scramble things up.

Avoid Simple Distributions

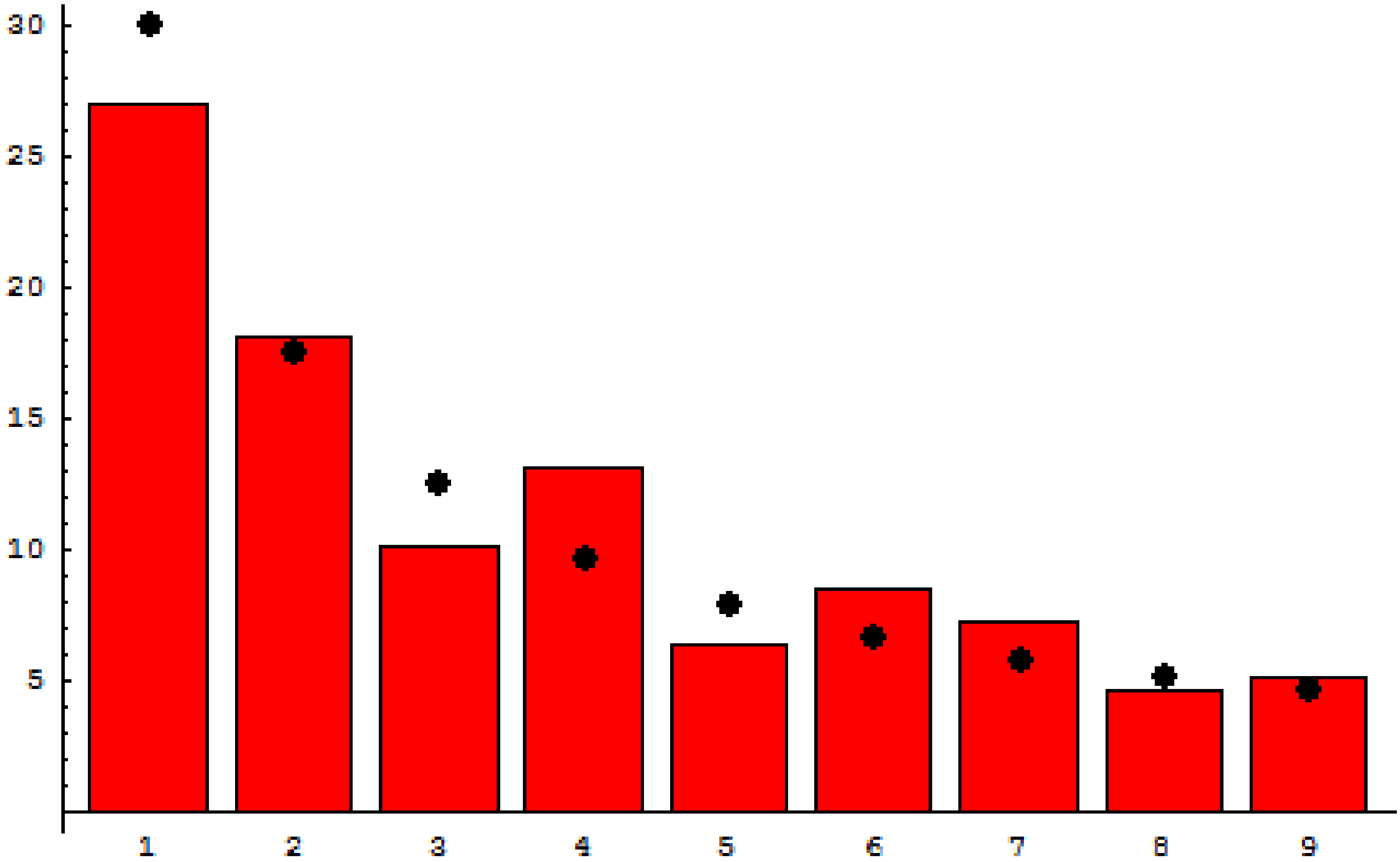
- Suppose you want to build a hash function for names.
- Earlier, we tried doing this by first letter.
- This is not a very good idea.

CS106B Name Distributions

By first letter of first name

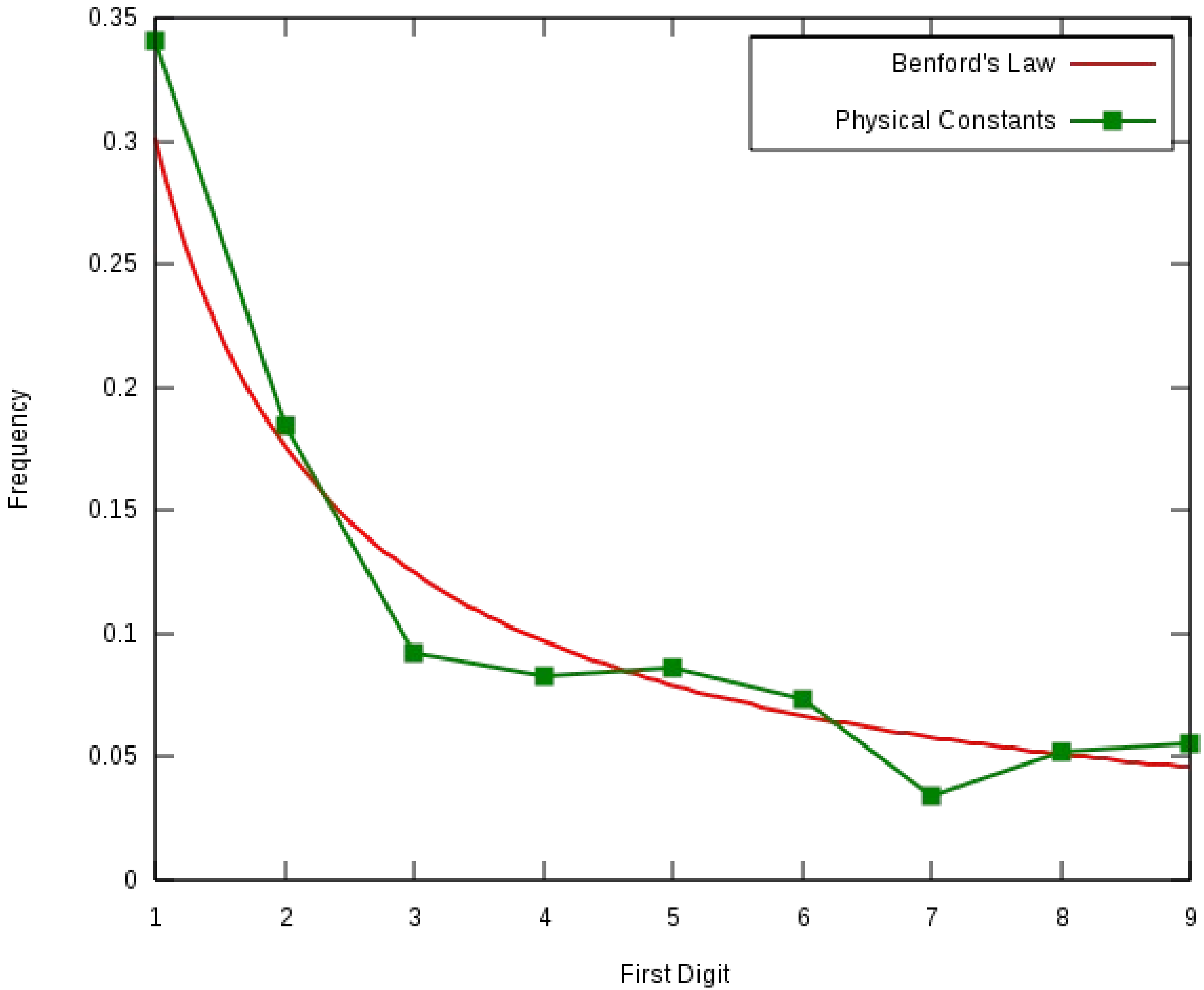


Benford's Law



http://en.wikipedia.org/wiki/File:Benfords_law_illustrated_by_world%27s_countries_population.png

Benford's Law



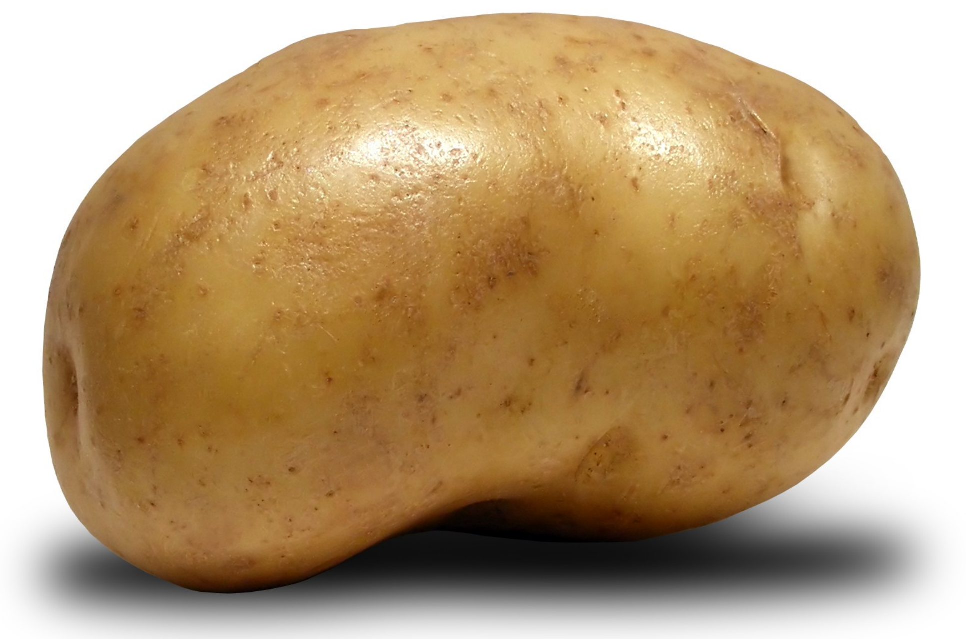
http://en.wikipedia.org/wiki/File:Benford-physical.svg

Building a Better Hash Function

- Designing good hash functions requires a level of mathematical sophistication far beyond the scope of this course.
  - Take CS161 for details!
- Generally, hash functions work as follows:
  - Scramble the input up in a way that converts it to a positive integer.
  - Using the `%` operator, wrap the value from a positive integer to something in the range of buckets.

Good Hash Functions

- A good hash function typically will scramble all of the bits of the input together in a way that appears totally random.
- Hence the name “hash function.”



Some Interesting Numbers

- For 300 students and 26 buckets, given an optimal distribution of names into buckets, an average of **5.77** lookups are needed.
- Using first letter of first name: an average of **9.56** lookups are needed.
- Using the SAX hash function: an average of **6.17** lookups are needed.
- That's 50% faster than by first letter!

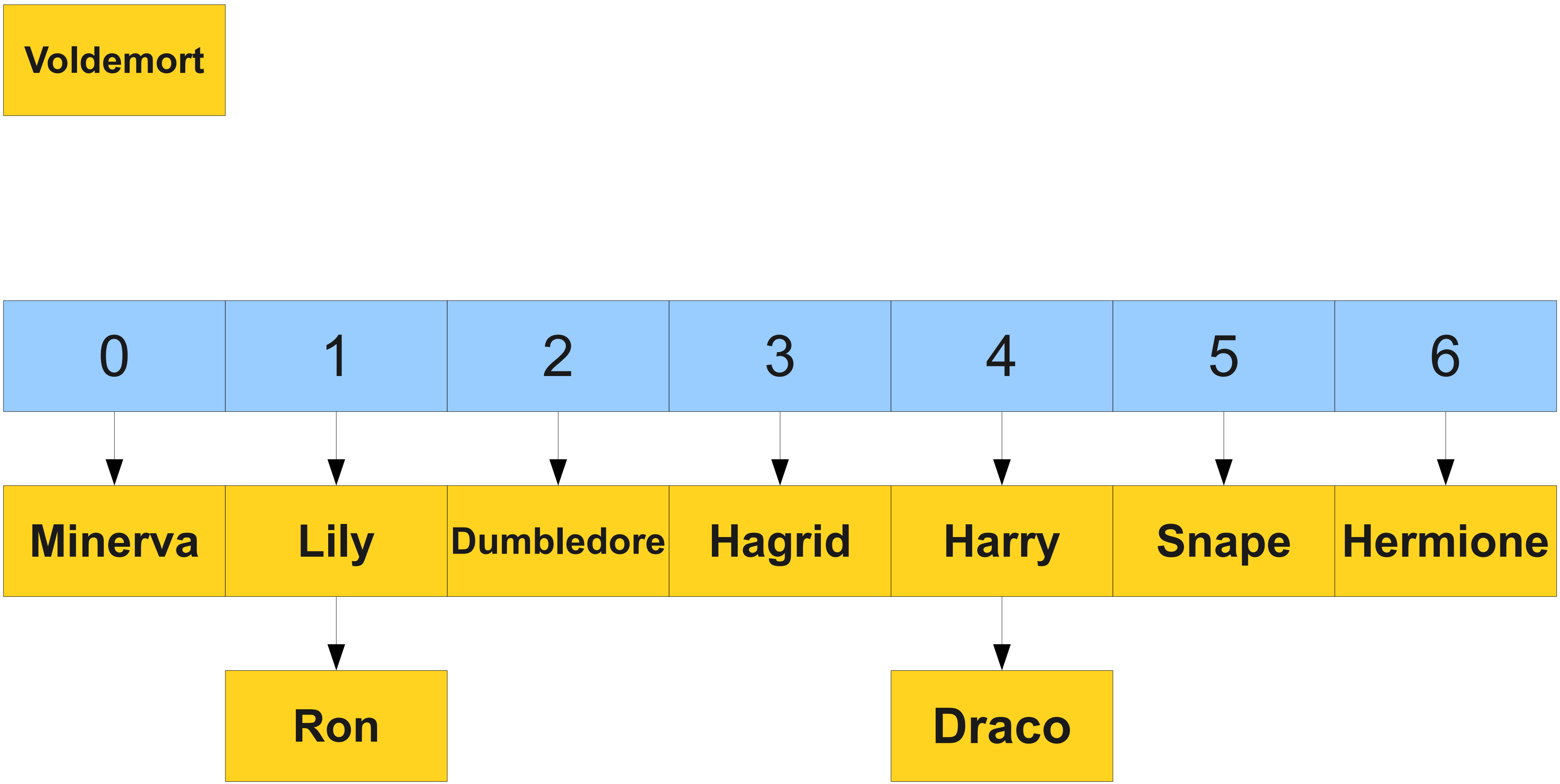
Hash Table Performance

- Suppose that we have n elements and m buckets.
- Assuming a good hash function, the expected time to look up an element is **$O(1 + n/m)$**.
- The ratio n/m is called the **load factor**.
- If we add buckets when the number of elements is large, we keep the load factor low.

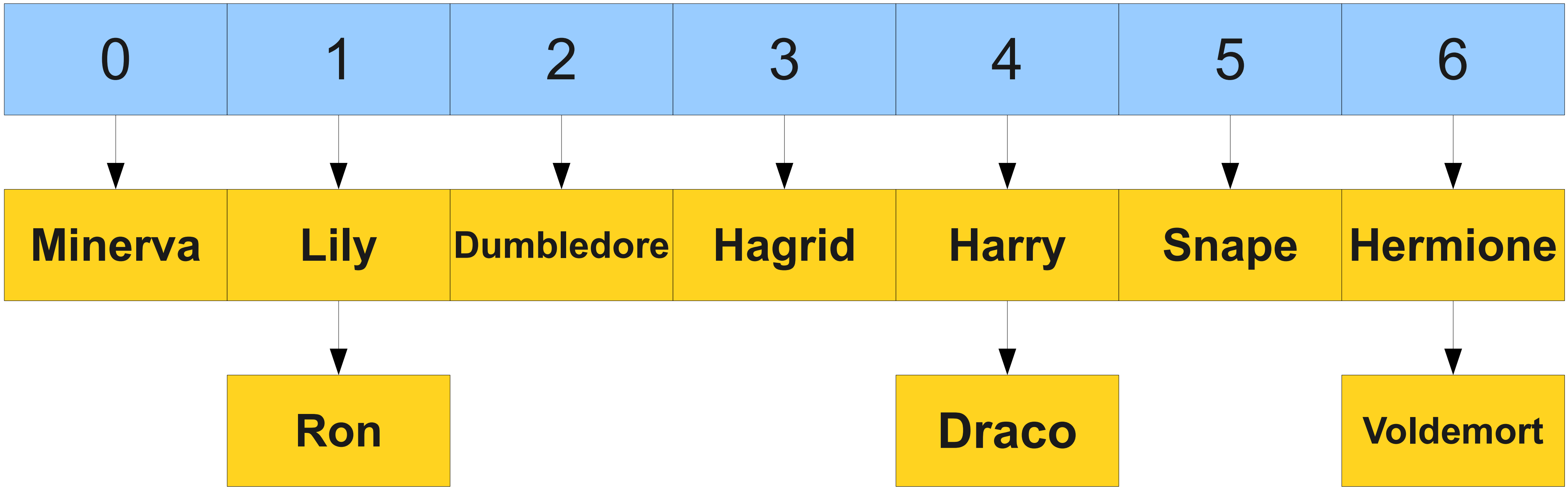
Hashing and Rehashing

Hashing and Rehashing

Hashing and Rehashing



Hashing and Rehashing



Hashing and Rehashing

- Idea: Track the number of buckets m and the number of total elements n.
- When inserting, if n / m exceeds some value (say, 2), double the number of buckets and redistribute the elements evenly.
- This makes $n / m \leq 2$, so the expected lookup time in a hash table is **O(1)**.

Putting it together: Building **`HashMap`**