Hashing

A Genomics Problem

- Suppose that you and I each own a genomics lab in which we store millions of human genomes.
  - Each genome is a six-billion character string.
- We want to compare which genomes we have in common and we have the ability to communicate over a network.
- Sending data over a network is ***much*** slower than processing the data locally.
  - Say, 1,000,000x slower.
- How might we determine which genomes we have in common?

A Naive Solution

- I send you all of my genomes and you compare them against the ones you have.
- **Pros:** Very easy to implement.
- **Cons:** *Extremely* slow.
  - Might have to transmit thousands of terabytes (*petabytes*) of information!
  - Even on a very fast network, this could take weeks.

A Slightly Better Solution

- I send you the first 1000 characters of each genome. (Remember a genome is six billion characters long).
- You look at the genomes you have that also start with that prefix and let me know which prefixes match.
- I then send you just those genomes, at which point you can find all matches.
- **Pros**: Cuts down data transmitted by a factor of *one million!*
- **Cons:** If many genomes start the same way, I might have to send you a *bunch* of redundant genomes.

Another Possible Solution

- In advance, we count up the number of each type of letter in each of our genomes. This gives a *frequency histogram*.
- I send you the frequency histograms for each of my genomes.
- You then let me know which histograms match your own histogram.
- I then send you the genomes matching those histograms. From there, you can find the matches.

Yet Another Possible Solution

- In advance, we run the following functions on each of our genomes:

```
string getSynopsis(string& input) {
  string result;
  for (int i = 0; i < input.size(); i += 1000000)
      result += input[i];
  return result;
}
```

- I send you the synopses of each of my genomes.
- You then let me know which of my synopses match your synopses.
- I then send you all genomes matching those synopses, from which you can find all matches.

The Essential Structure

- The general sketch of these latter approaches is:
  - In advance, we find some quick way of summarizing our genomes.
  - I send you just the summaries.
  - You find genomes that match the summaries and let me know which ones match.
  - I only send you complete genomes over the network if this first step yields a match.
- I might send you *more* genomes than you need, but I will never send you *fewer* genomes than you need.

The Essential Structure

The general sketch of these latter approaches is:

- In advance, we find some quick way of summarizing our genomes.

  I send you just the summaries.

  You find genomes that match the summaries and let me know which ones match.

  I only send you complete genomes over the network if this first step yields a match.

I might send you *more* genomes than you need, but I will never send you *fewer* genomes than you need.

Hash Functions

- A **hash function** is a function that converts a large object (a genome, a string, a sequence of elements, etc.) into a smaller object (a shorter string, an integer, etc.)
- A hash function ***must*** be deterministic: given an input, it must always produce the same output.
  - *Why?*
- A hash function ***should*** try to produce different outputs for different inputs.
  - Not always possible if there are only finitely many possible outputs.

Why Hash Functions Matter

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`** efficiently?
- There are many options; we'll see one today.

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?
  - Sorted: **O(*n*)**.
  - Unsorted: **O(*n*)**.
- What is the big-O time complexity of looking up a key?
  - Sorted: **O(log *n*)**.
  - Unsorted: **O(*n*)**.

Knowing Where to Look

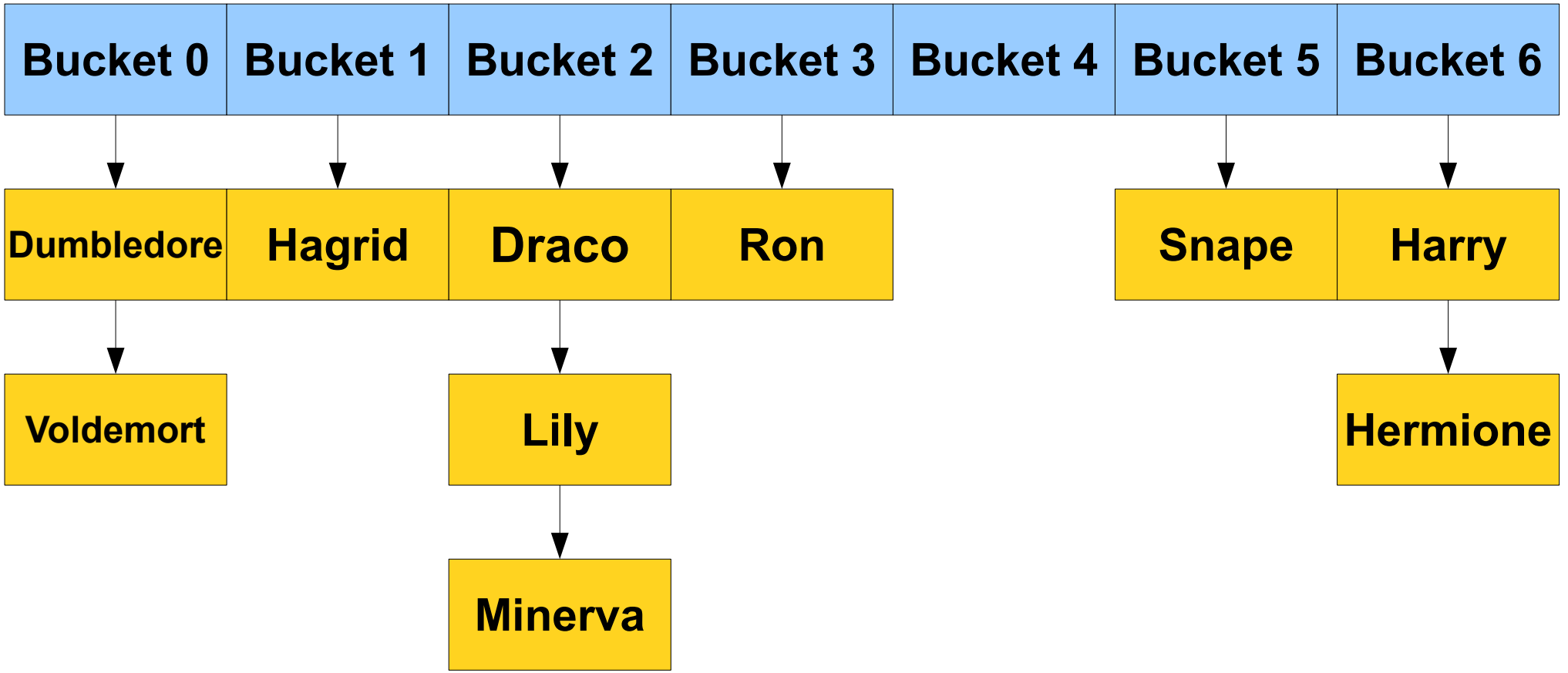
- Our linked-list **`Queue`** implementation has O(1) enqueue, dequeue, and front.
- Why is this?
- Know exactly where to look to find or insert a value.
- **`Queue`** implementation was O(n) for enqueue, but was improved to O(1) by adding extra information about where to insert.

An Example: Clothes

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



Why Linked Lists?

- A dynamically allocated array of linked lists!
- This seems complicated, why are we using linked lists instead of **`Vector`**s?
  - We'll give a very good reason for doing this.

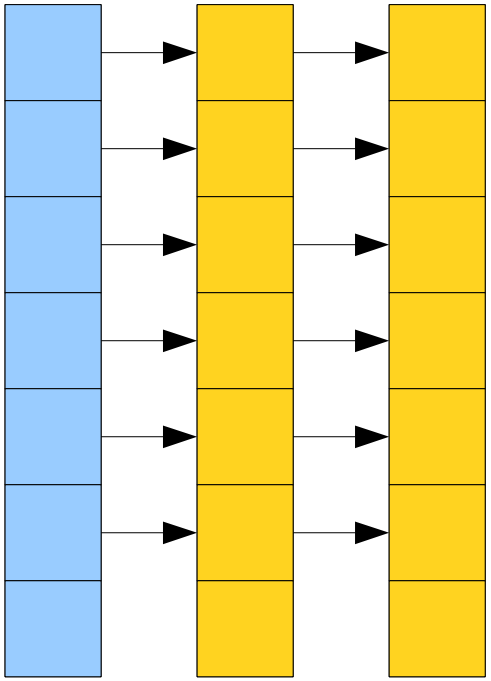
How Do We Distribute Elements?

- **Use a hash function!**
  - The input to the hash function is the object to distribute.
  - The output of the function is the index of the bucket in which it should be.
- To do a lookup:
  - Apply the hash function to the object to determine which bucket it belongs to.
  - Look at all elements in the bucket to determine whether it's there.
- This data structure is called a **hash table**.

```
OurHashMap::OurHashMap()
OurHashMap::~OurHashMap()
```

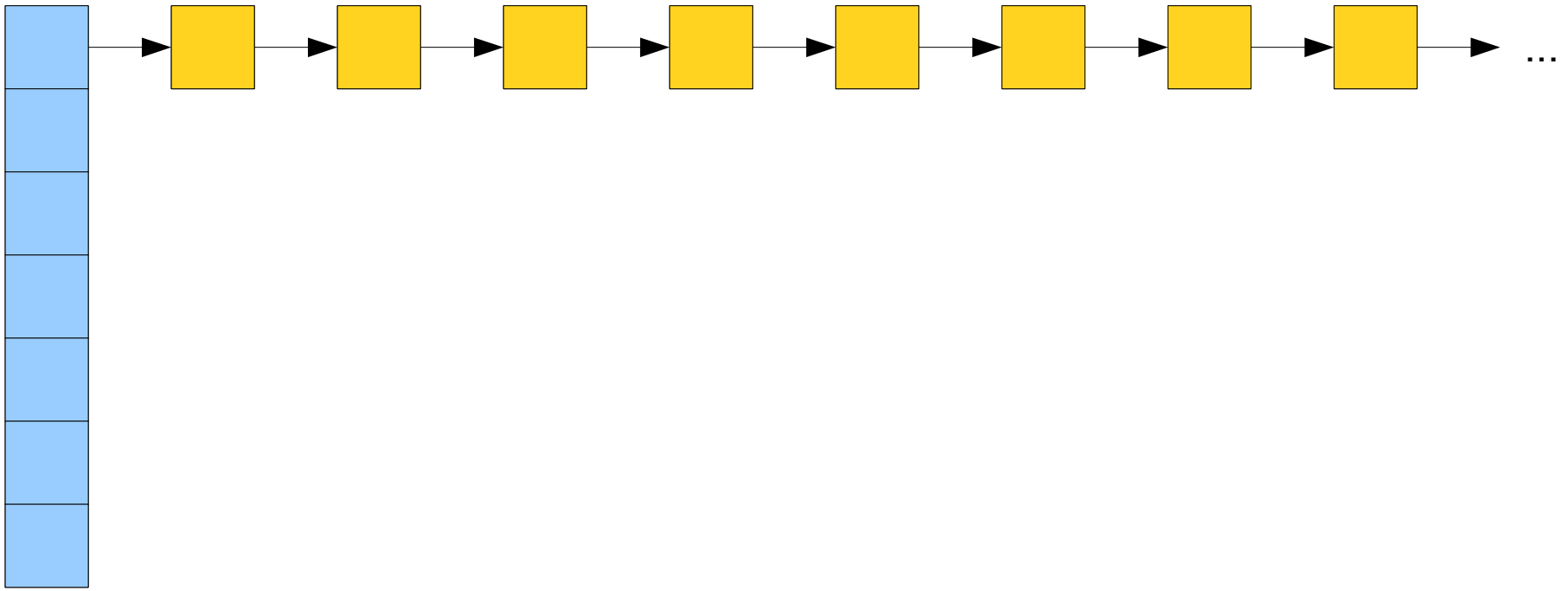
Distributing Keys

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

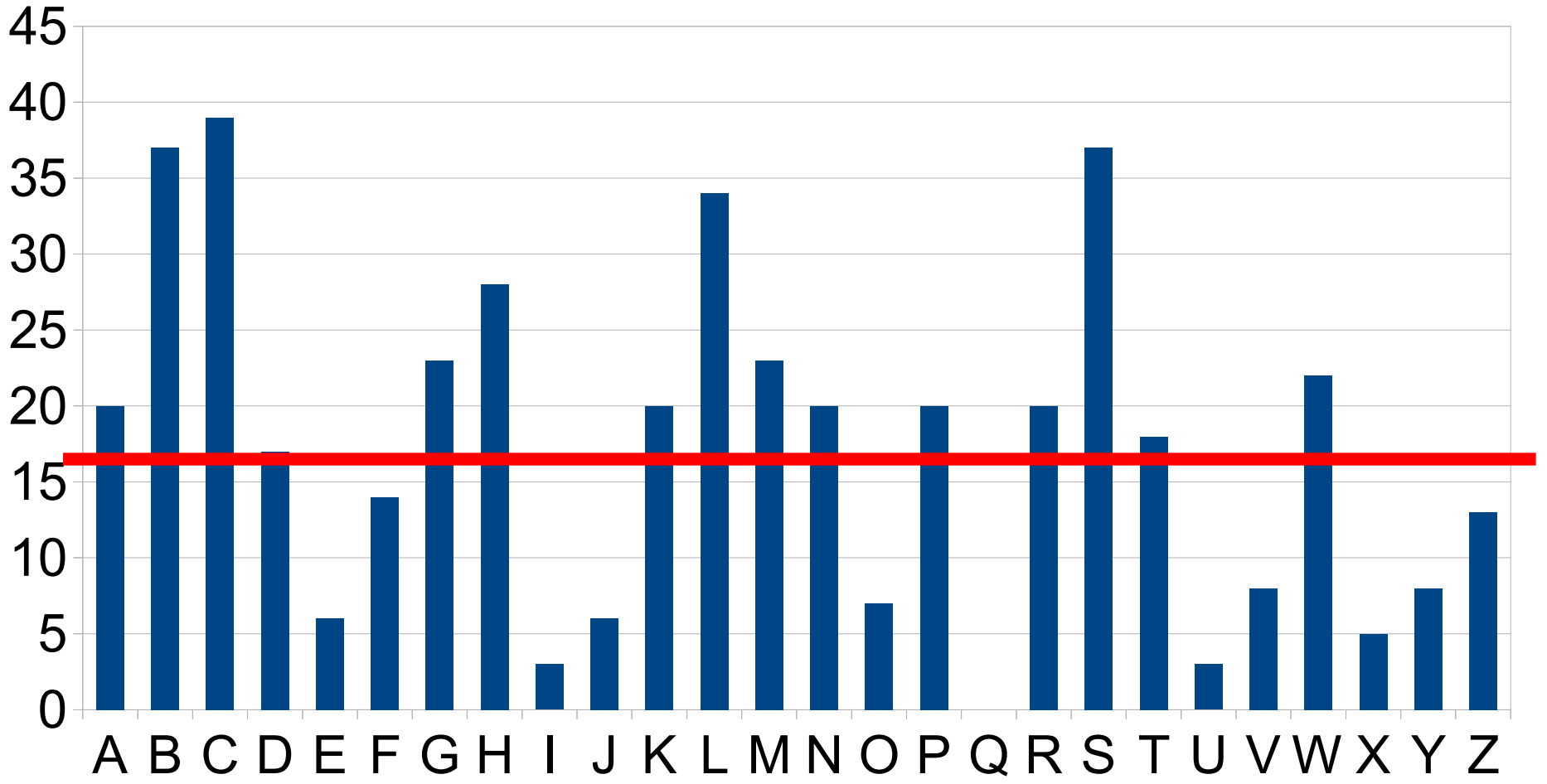


Distributing Keys

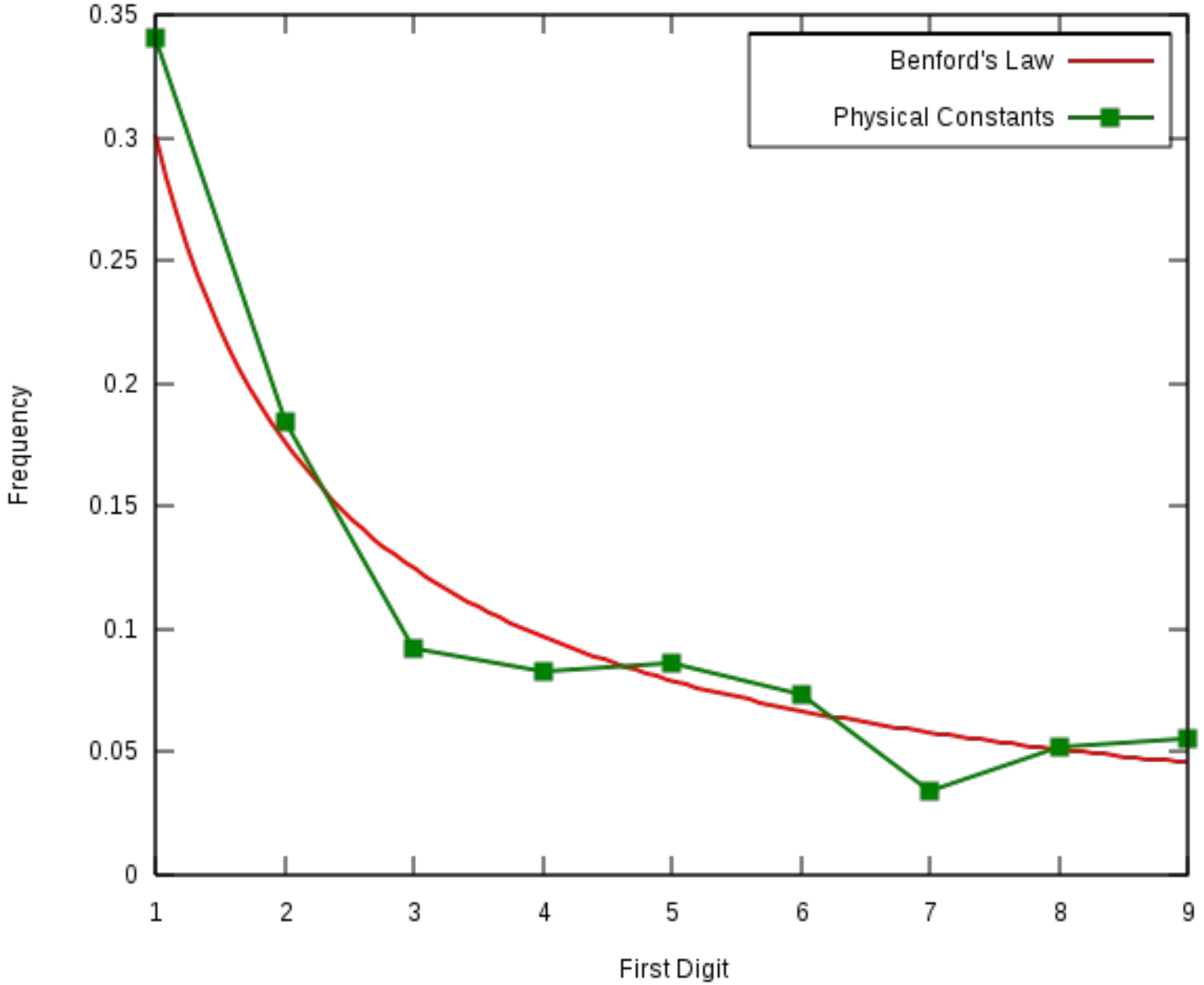
- We want to choose a hash function that will distribute elements as evenly as possible to try to guarantee a nice, even spread.
- Suppose you want to build a hash function for names.
- One initial idea: Hash each last name to the first letter of that last name.
- How well will this distribute elements?

Spring CS106B Name Distributions

By First Letter of Last Name



Benford's Law

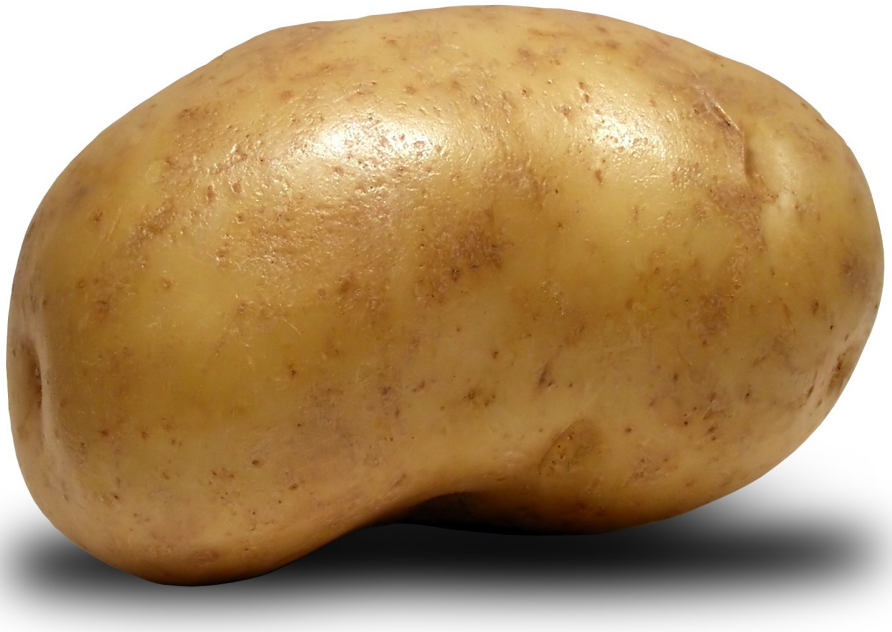


Building a Better Hash Function

- 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.”



Bad Hash Functions

Bad Hash Functions #1

```
int myHash(string key) {
   return 0;
}
```

Bad Hash Functions #1

```
int myHash(string key) {
   return 0;
}
```

All key will be put in the same bucket!

Bad Hash Functions #2

```
int myHash(string key) {
   return randomInteger(0,NUM_BUCKETS);
}
```

Bad Hash Functions #2

```
int myHash(string key) {
   return randomInteger(0,NUM_BUCKETS);
}
```

Can't look up elements!

Bad Hash Functions #3

```
int myHash(string key) {
   int sum = 0;
   for (int i = 0; i < key.length(); i++) {
      sum += key[i];
   }
   return sum;
}
```

Bad Hash Functions #3

```
int myHash(string key) {
   int sum = 0;
   for (int i = 0; i < key.length(); i++) {
      sum += key[i];
   }
   return sum;
}
```

All permutations of the same string will be put in the same bucket!

myHash(“abc”) = myHash(“cab”)

test-hash-codes.cpp

Some Interesting Numbers

- For 451 students and 26 buckets, given an optimal distribution of names into buckets, an average of **8.65** lookups are needed.
- Using first letter of first name: an average of **12.7** lookups are needed.
- Using the SAX hash function: an average of **9.6** lookups are needed.
- That's 25% faster than by first letter!

OurHashMap::put()
OurHashMap::get()

Hash Table Performance

- Suppose that we have n elements and b buckets.
- Assuming a good hash function, the expected time to look up an element is **$O(1 + n / b)$**.
- The ratio n / b is called the **load factor**.
- Intuitively, this makes sense – if the elements are distributed evenly, you only need to look, on average, at n / b of them.

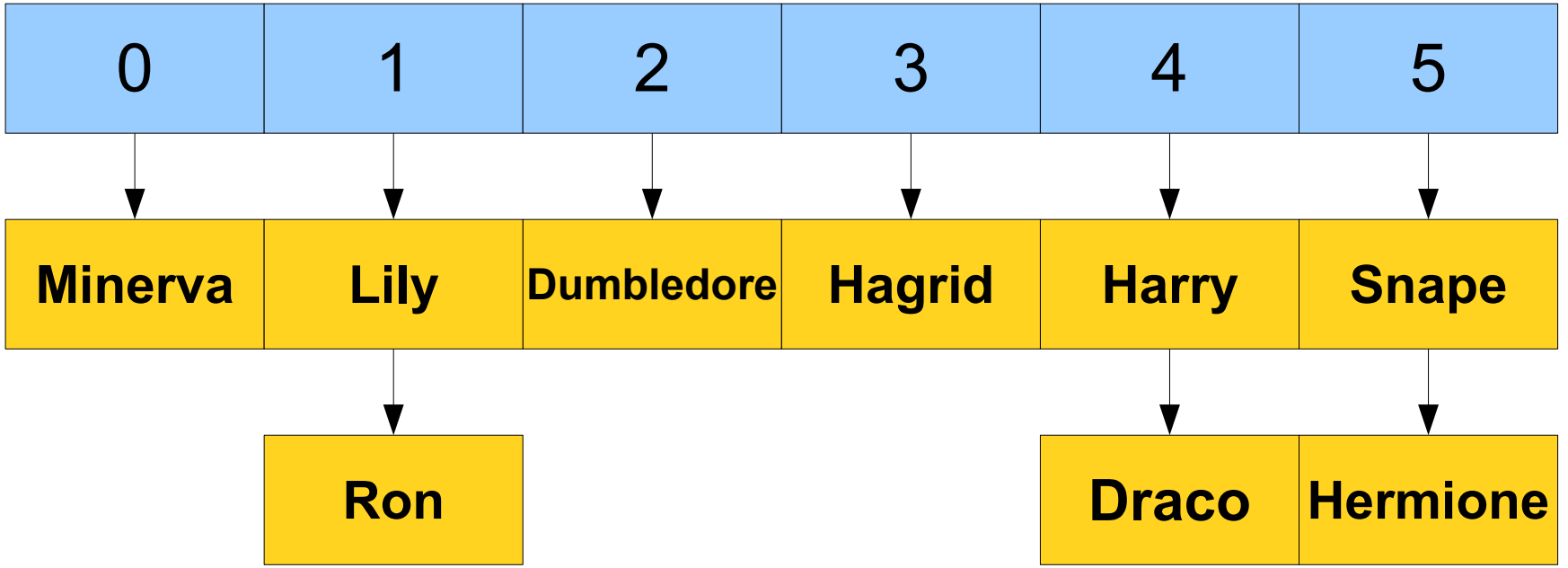
Hashing and Rehashing

Hashing and Rehashing

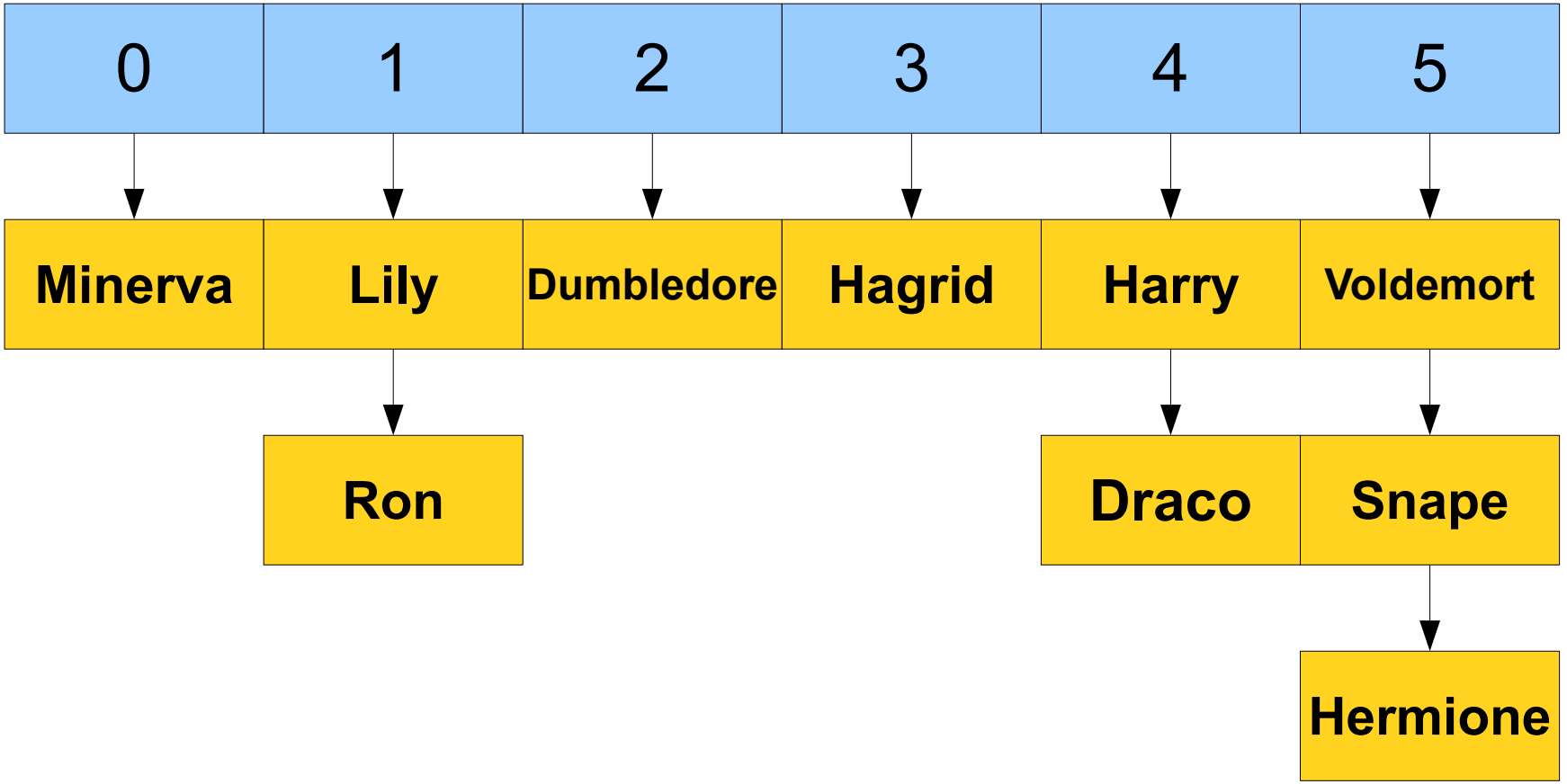
Voldemort

Hashing and Rehashing

Voldemort

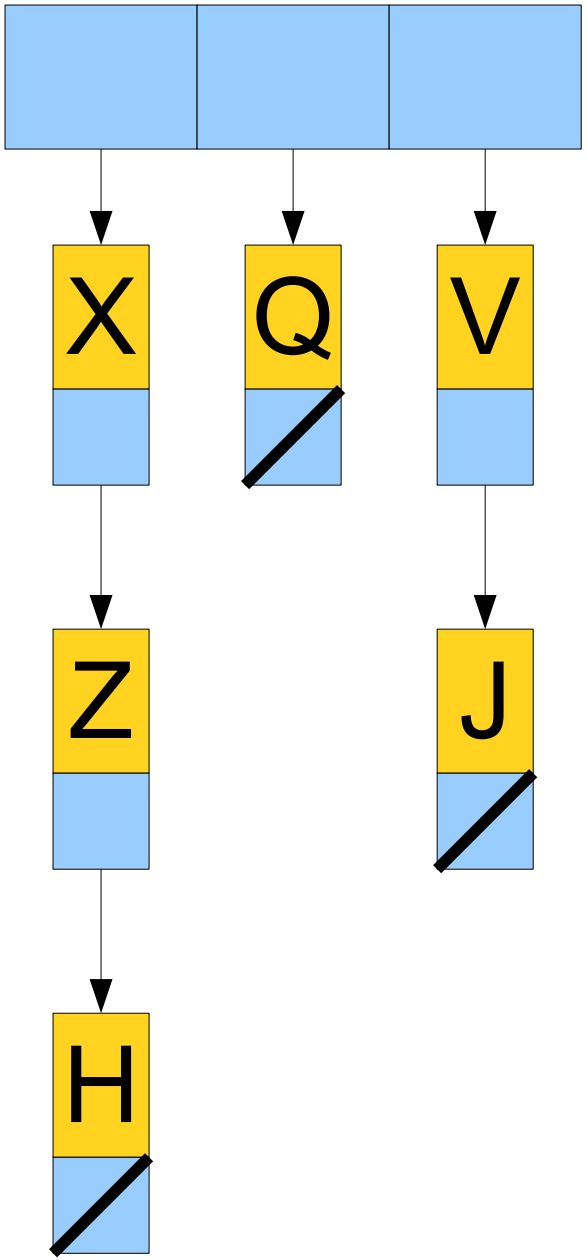


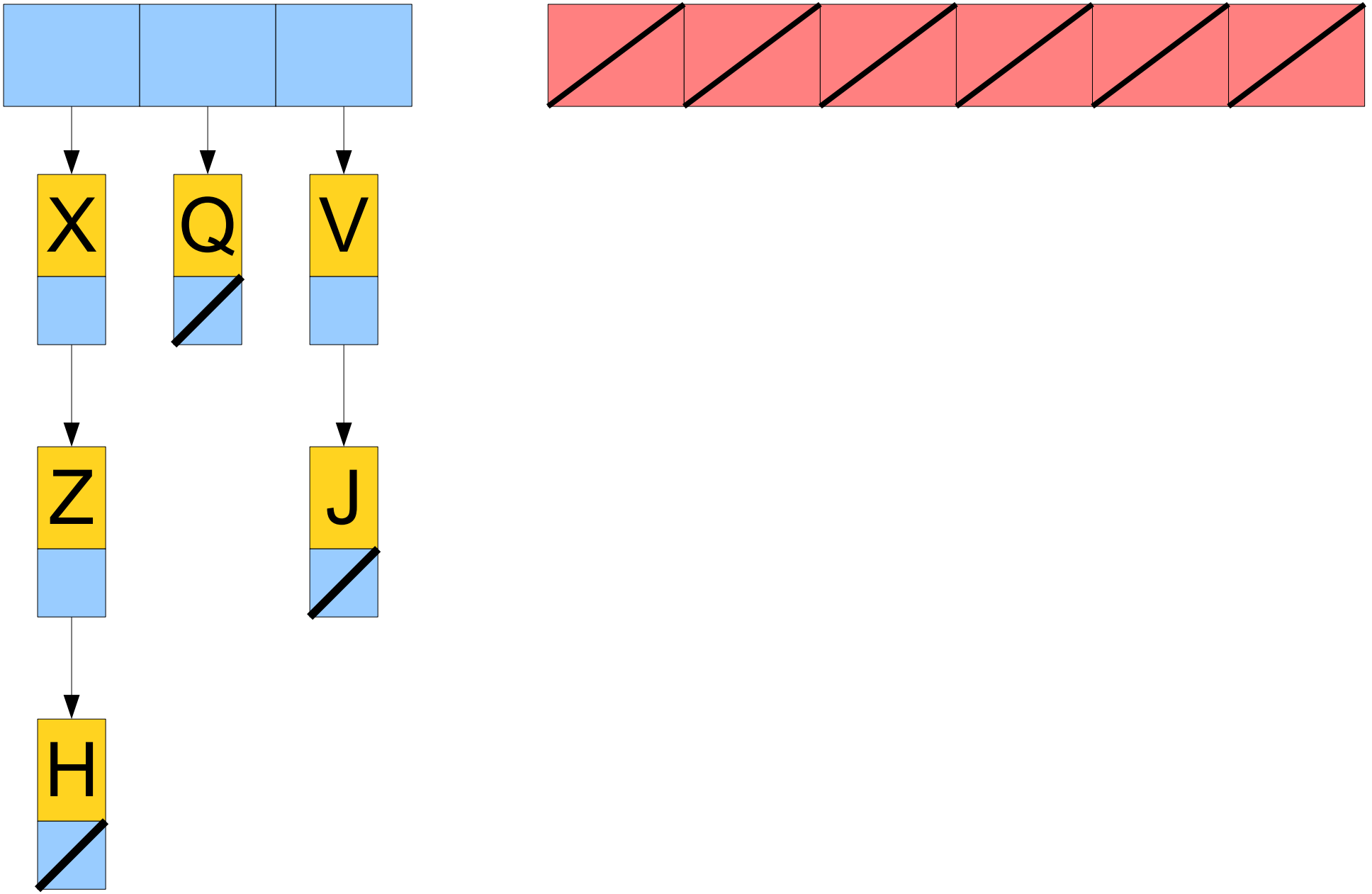
Hashing and Rehashing

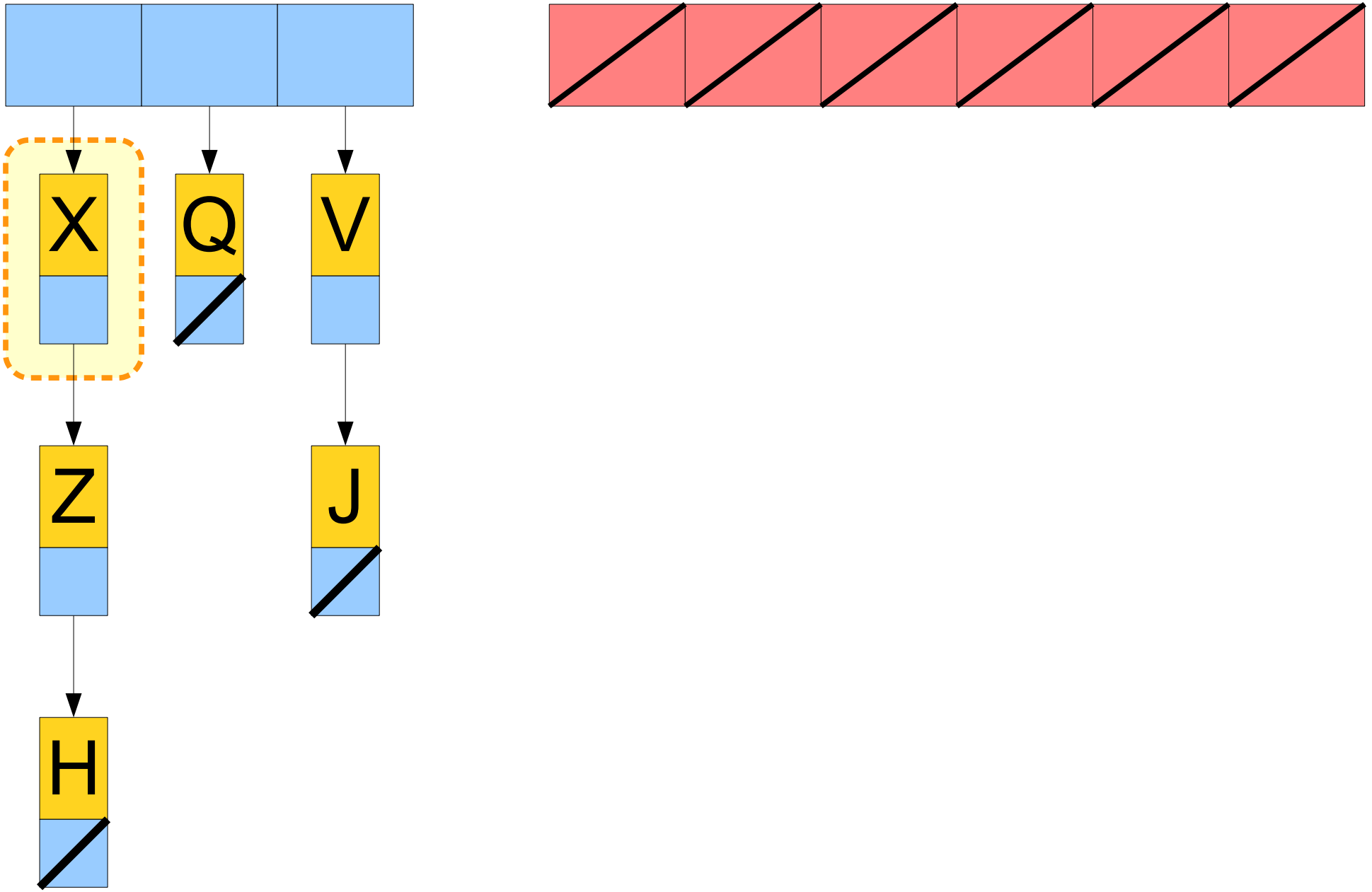


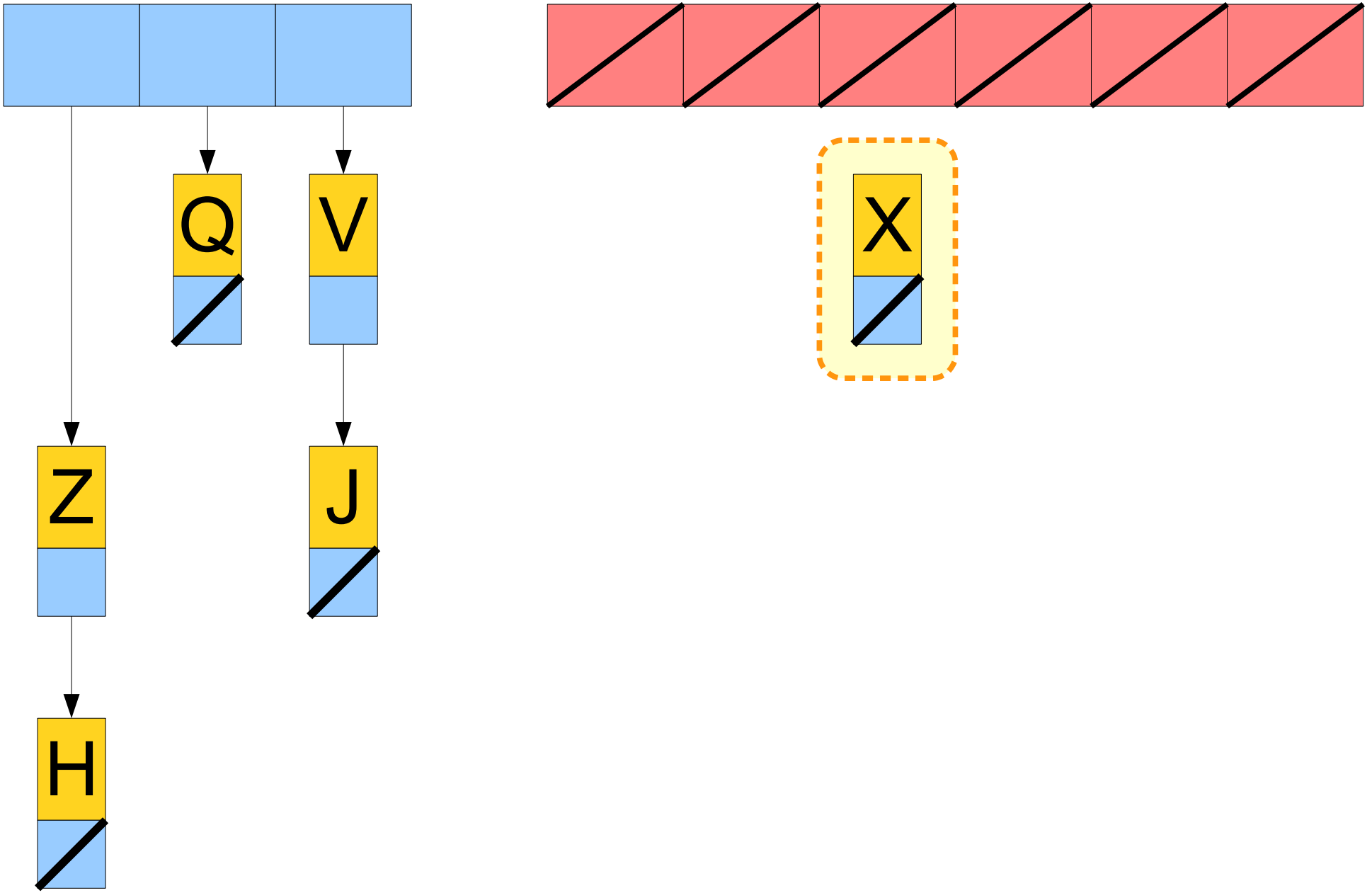
Hashing and Rehashing

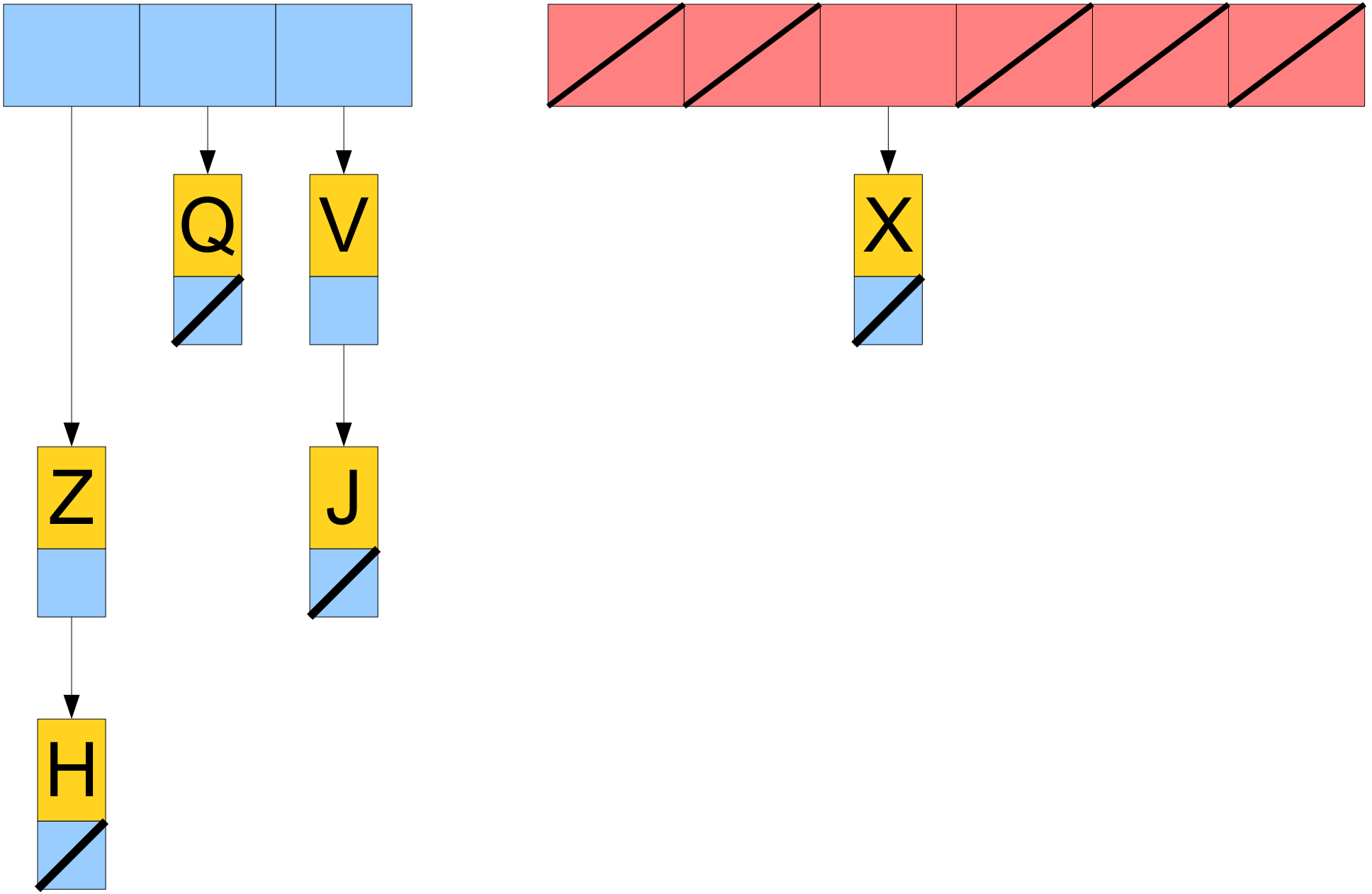
- Idea: Track the number of buckets b and the number of total elements n.
- When inserting, if n/b exceeds some small constant (say, 2), double the number of buckets and redistribute the elements evenly.
- This makes $n/b \leq 2$, so the expected lookup time in a hash table is **O(1)**.
- On average,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

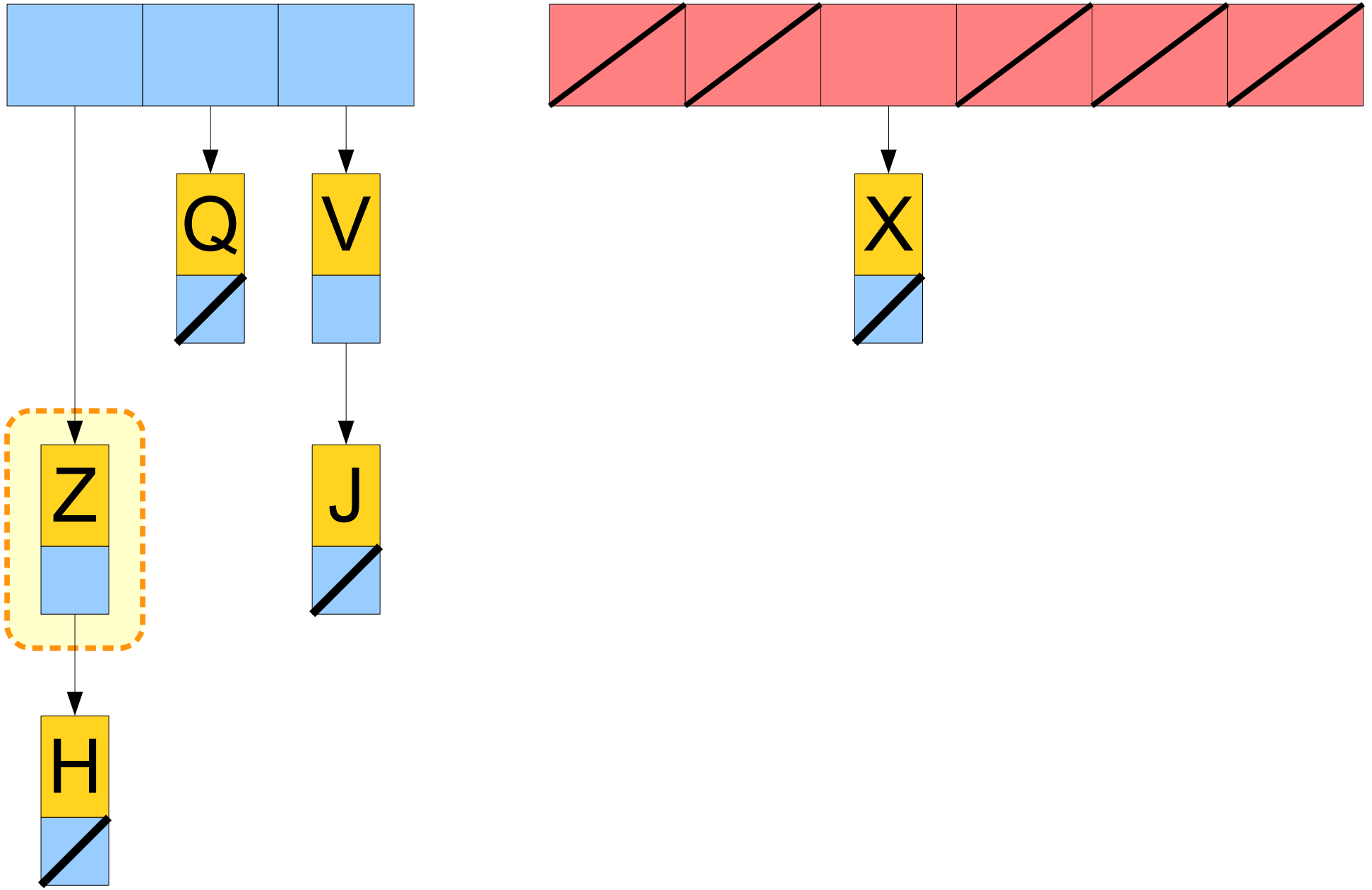

X
Q
V
Z
J
H

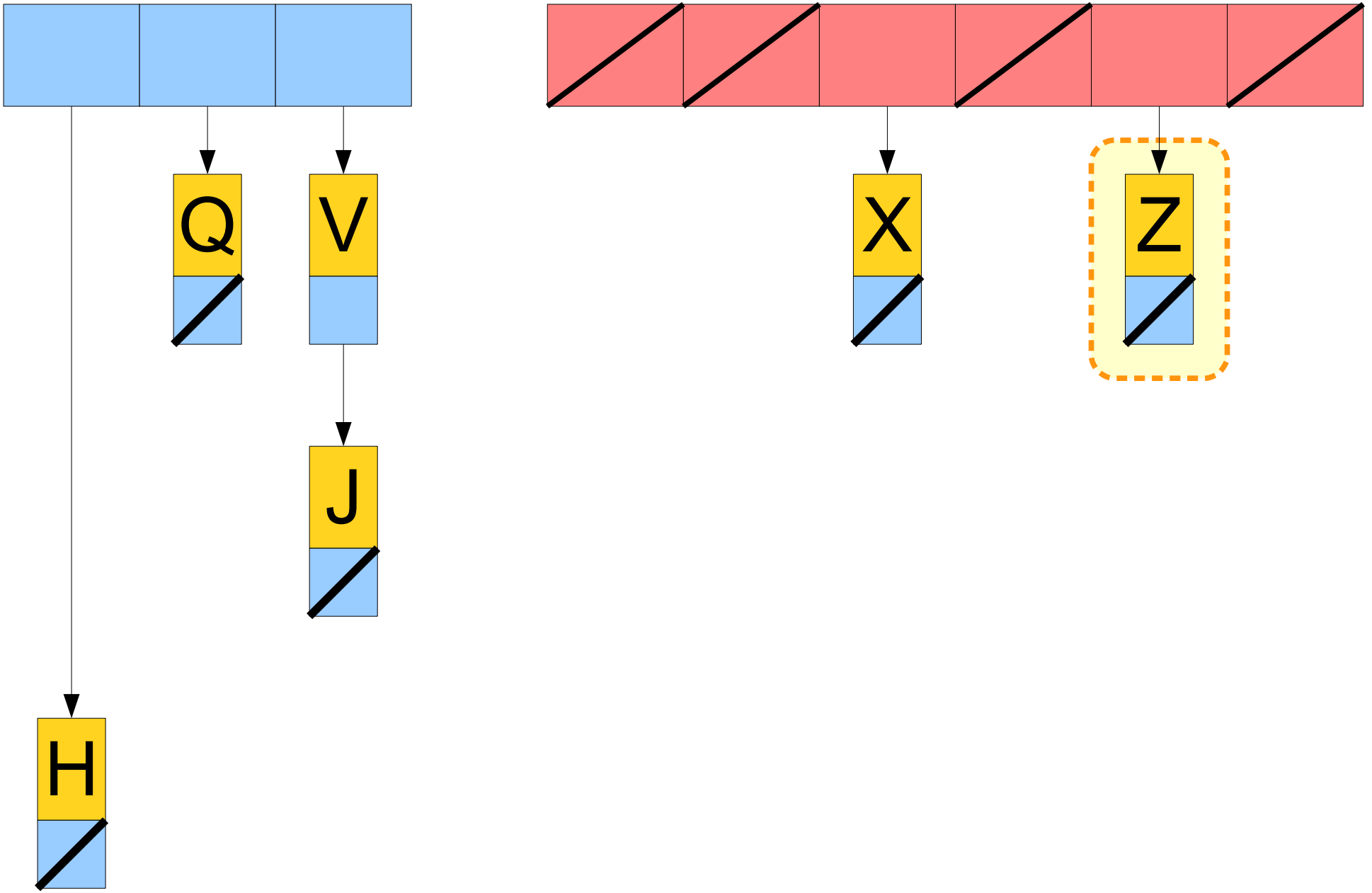

X
Q
V
Z
J
H

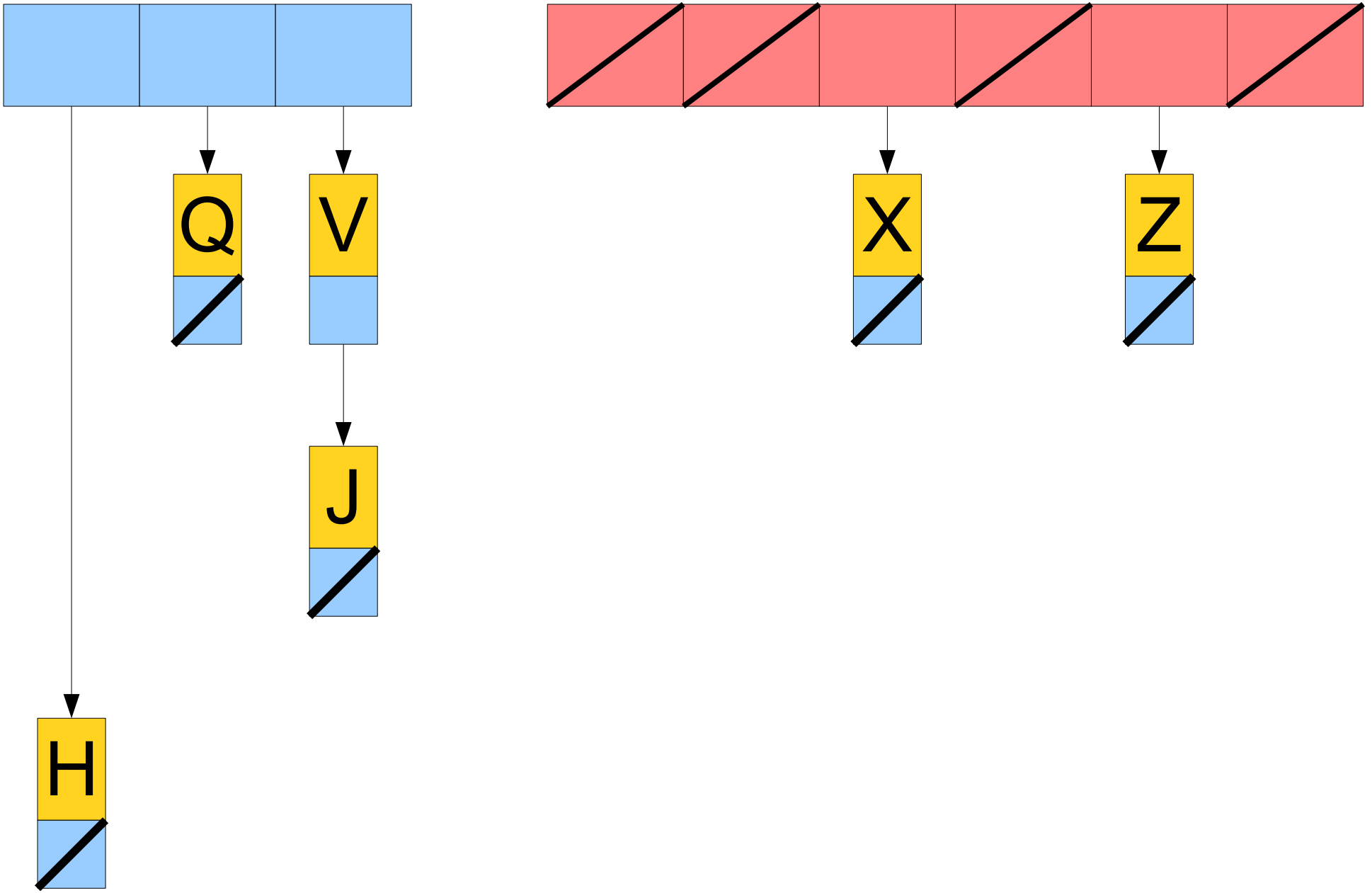

X
Q
V
Z
J
H

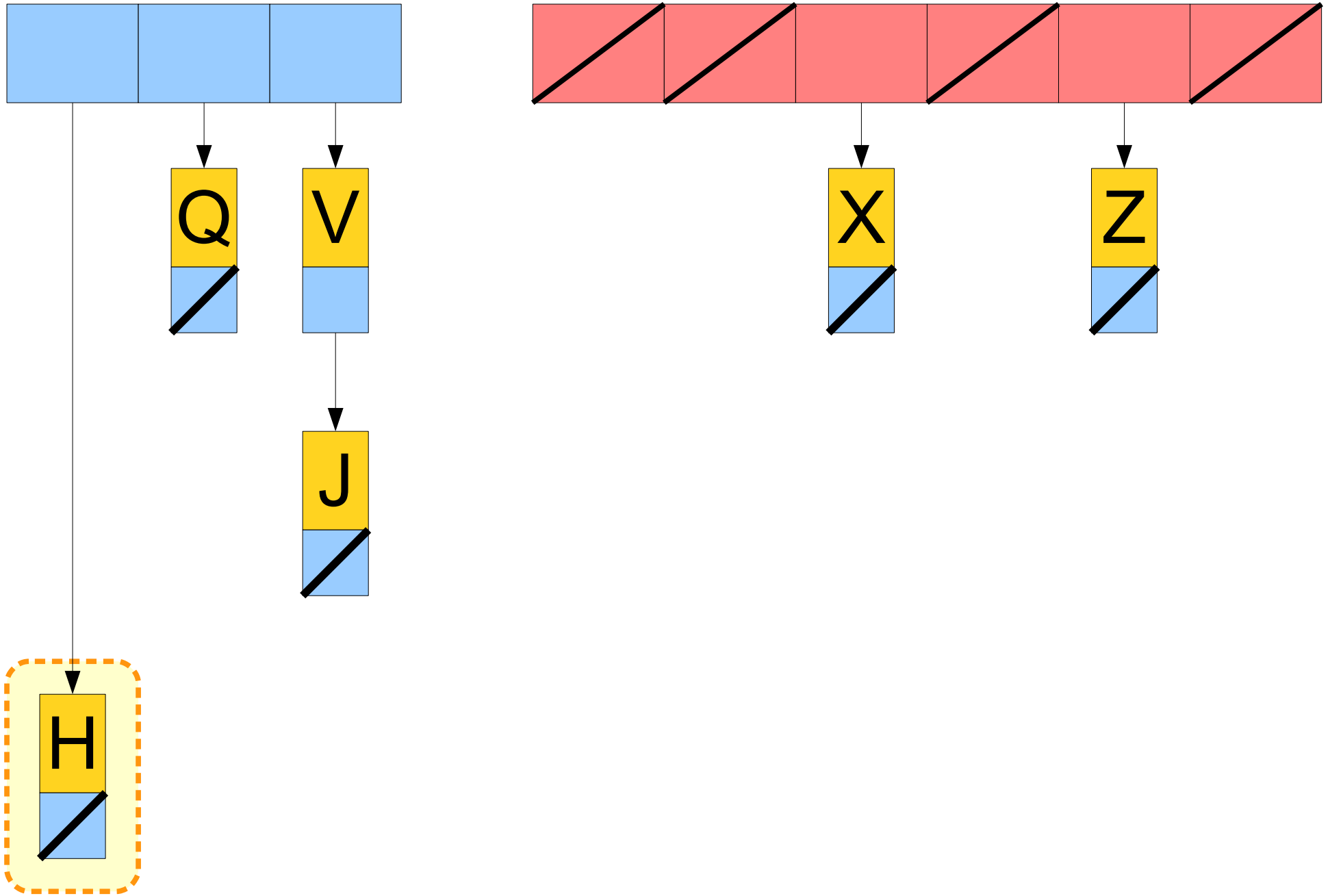

Q
V
X
Z
J
H

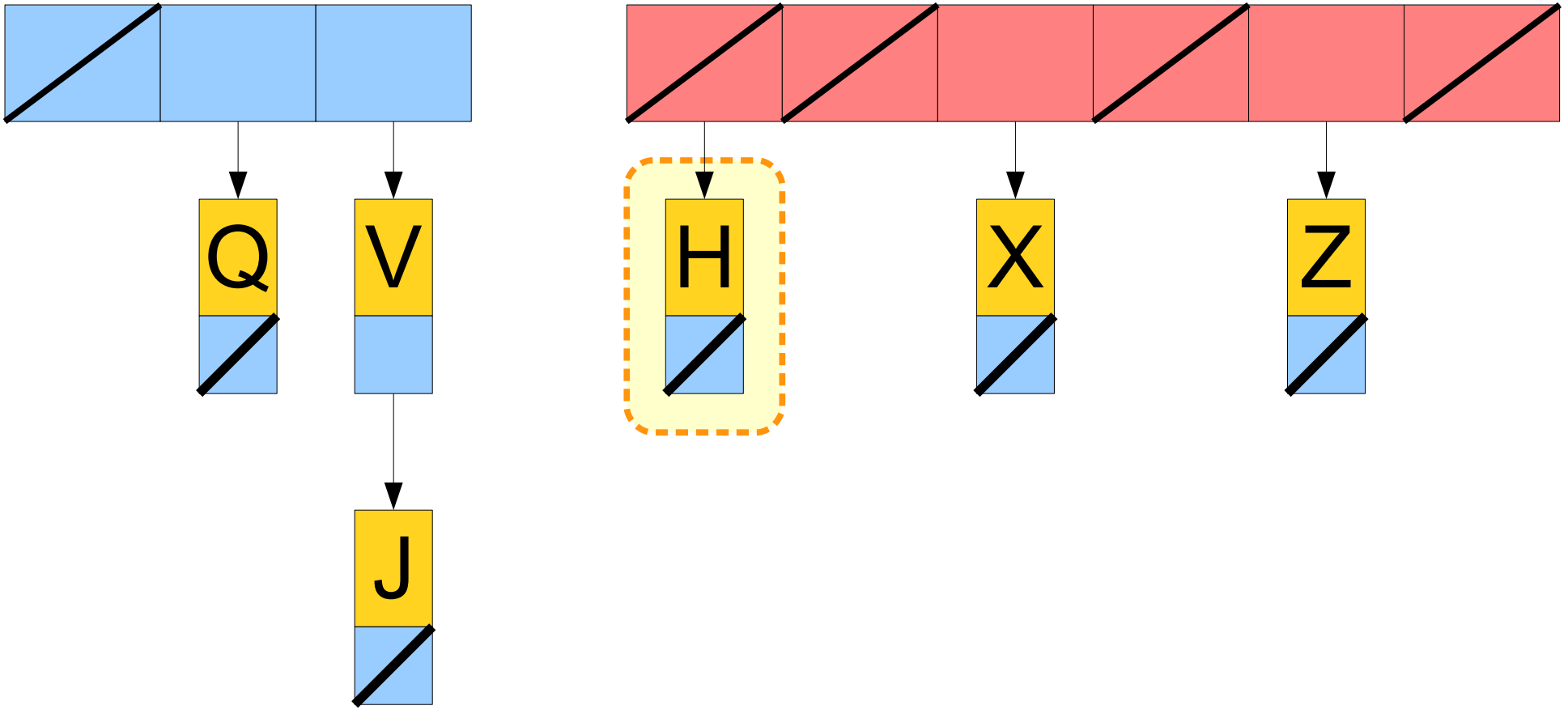

Q
V
X
Z
J
H

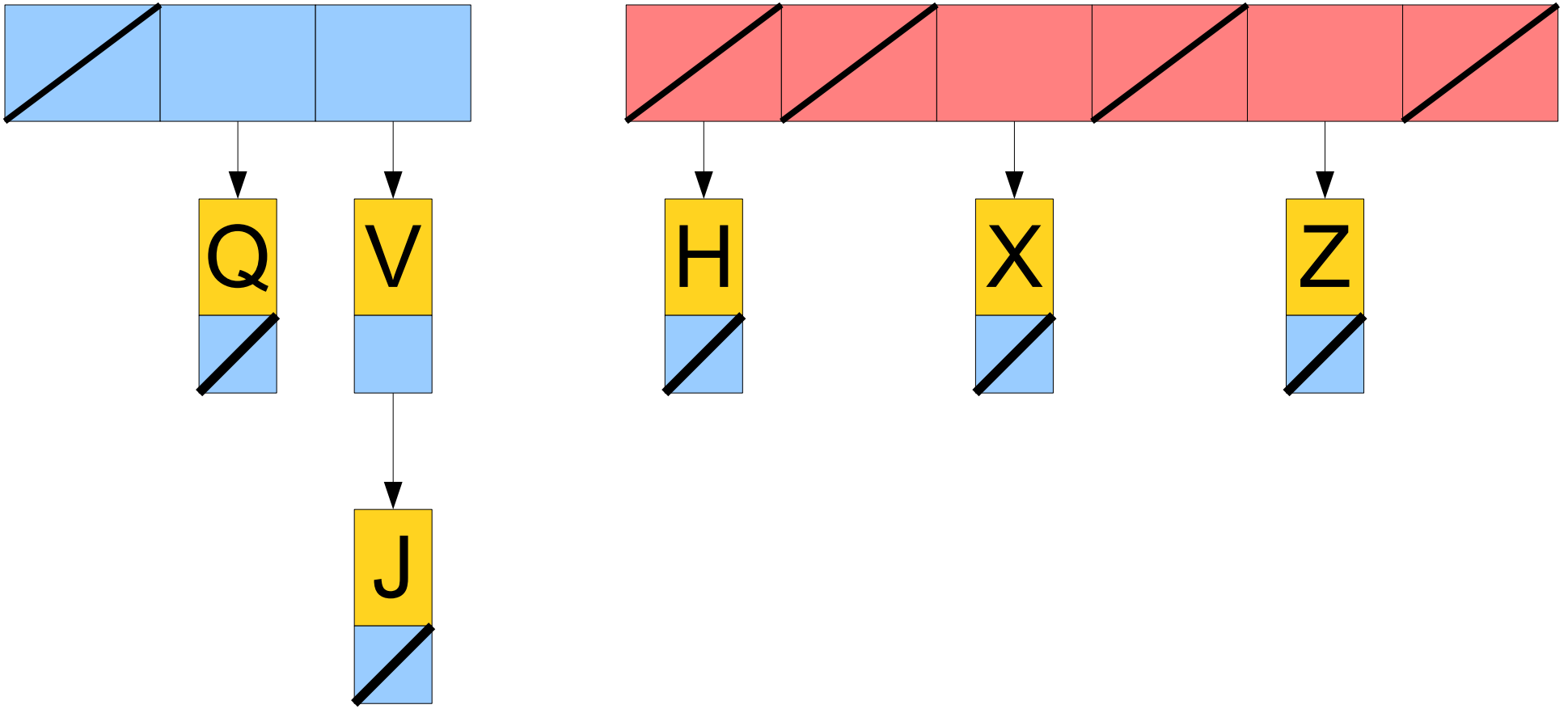

Q
V
X
Z
J
H


Q
V
J
H
X
Z


Q
V
J
H
X
Z


Q
V
J
H
X
Z


Q
V
J
H
X
Z


Q
V
J
H
X
Z

Why Linked Lists?

- Because we use linked lists, we don't need to create a bunch of new **`Vector`**s when we rehash!

`OurHashMap::rehash()`

The Final Analysis

- Expected time to do a lookup: **O(1)**.
- Expected time to do an insertion:
  - Every n elements, must double the table size and rehash. Does O(n) work, but only every n iterations.
  - Then does O(1) expected work to do the insertion.
  - **Amortized expected O(1) insertion!**

Next Time

- **Binary Search Trees**
  - Why are our `Map` and `Set` stored in sorted order?