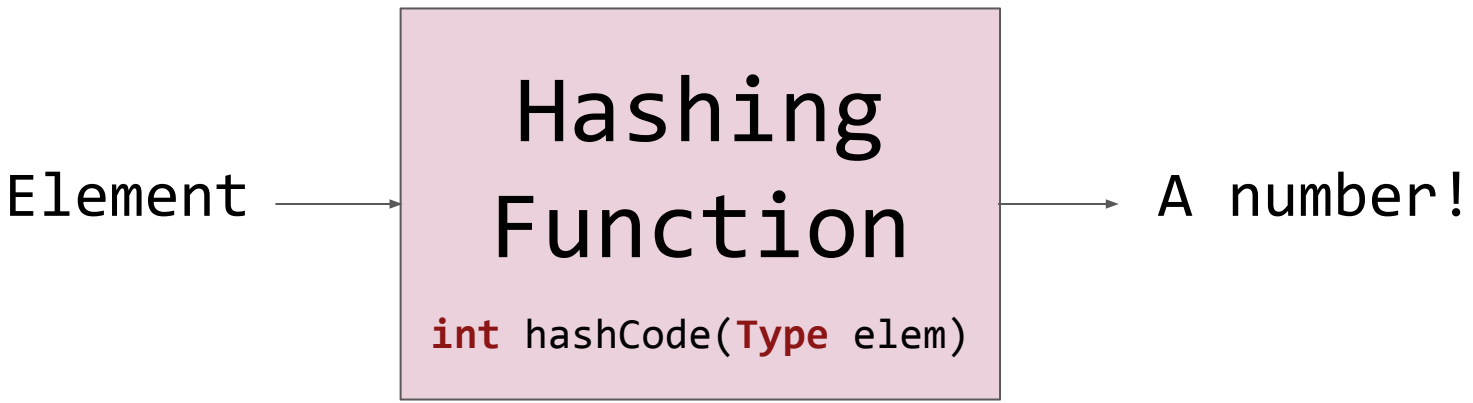
Lecture 22 - Hashing

CS106B - Monday, May 22, 2017

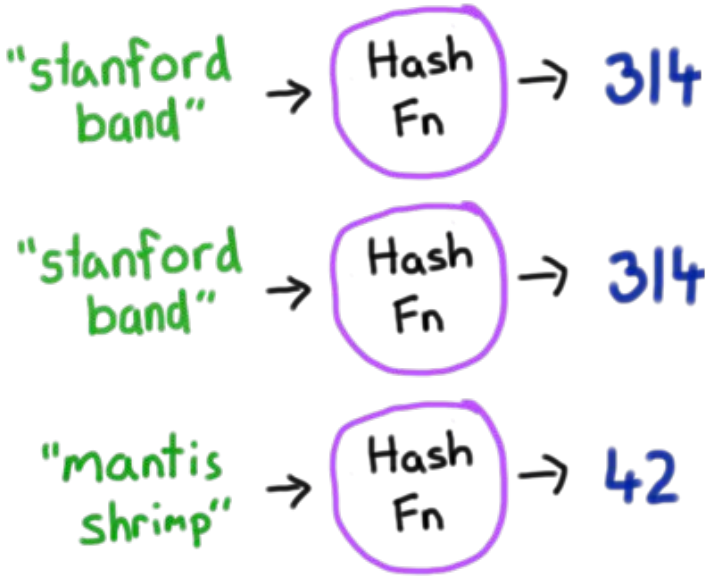
Anton Apostolatos

This presentation is based on lectures given by Chris Gregg, Chris Piech and Keith Schwarz.


Element
Hashing Function
int hashCode(Type elem)
A number!

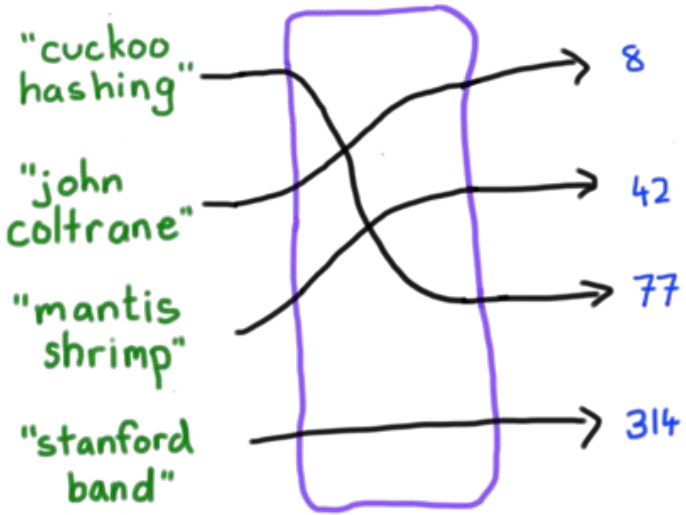
Property 1 - Deterministic

If you pass in the same input, you will **always** get the same output



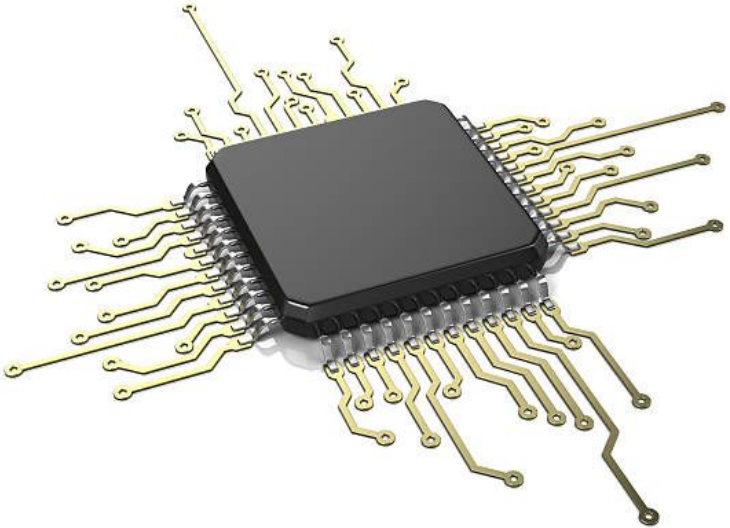
Property 2 - Well-Distributed

The numbers produced are as spread out as possible



Property 3 - Efficient and quick

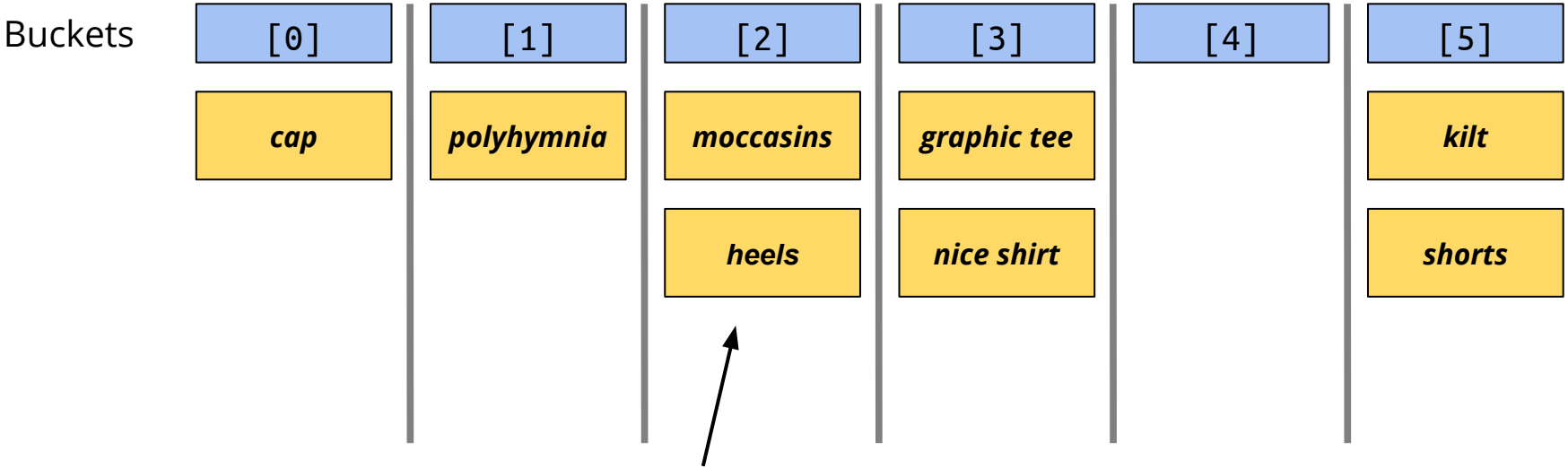
The hash function need to run *quickly*



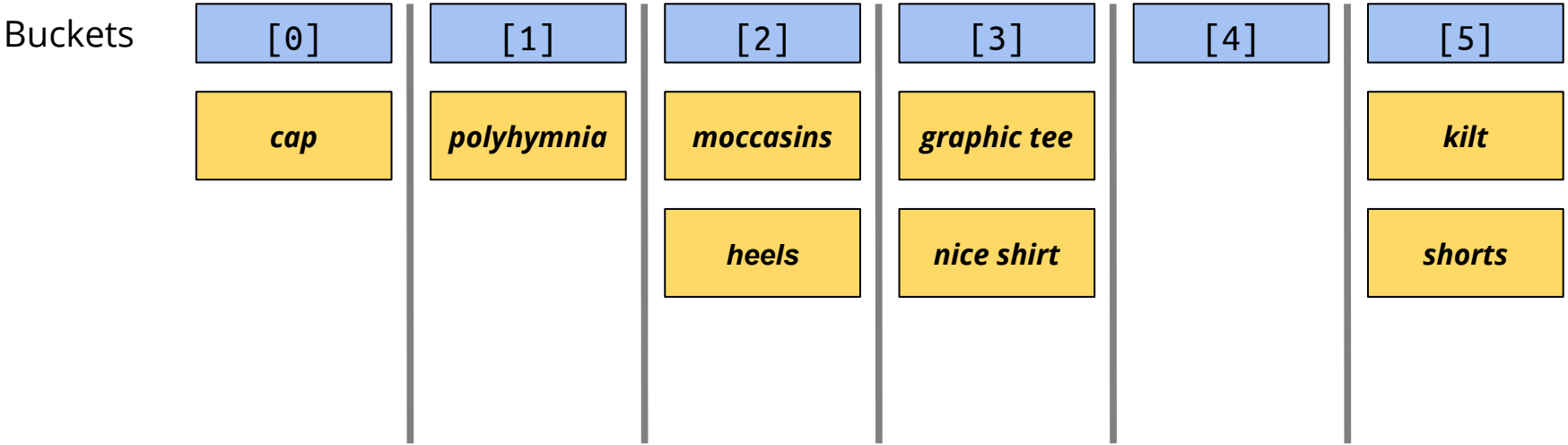
An example of why hashing is such a powerful tool…

Our strategy

- Maintain a small number of collections called ***buckets*** (think drawers)
- Find a ***rule*** that tells us where each object should go (knowing which drawer something should go to)
- To find something, ***only*** look at the bucket assigned to it (looking for a sock in the sock compartment)

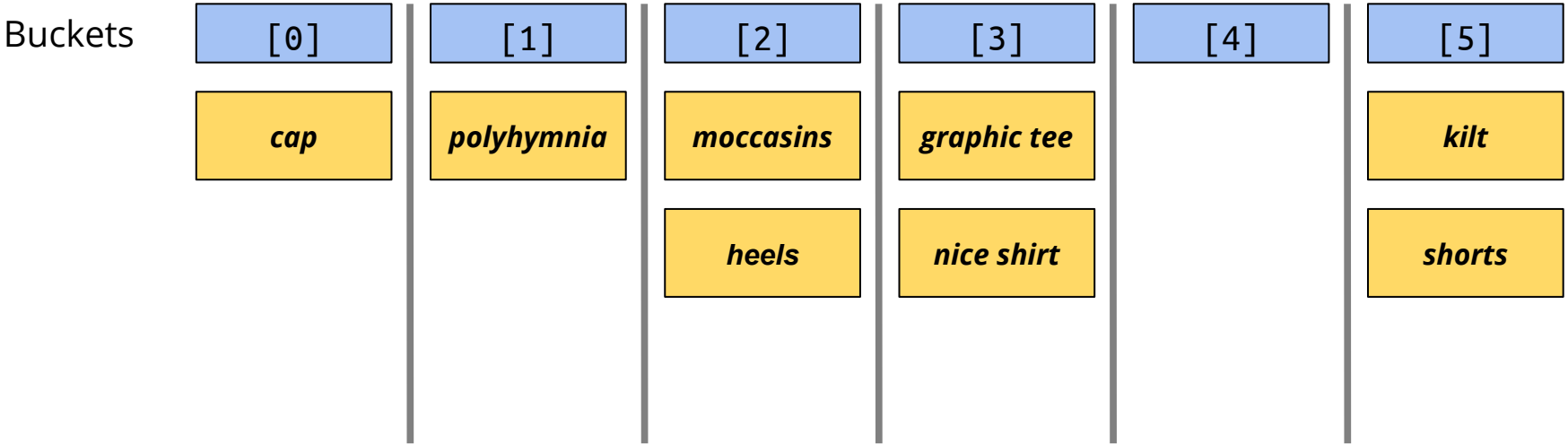


Linked List or **Vector**



Don't forget our nifty tool:

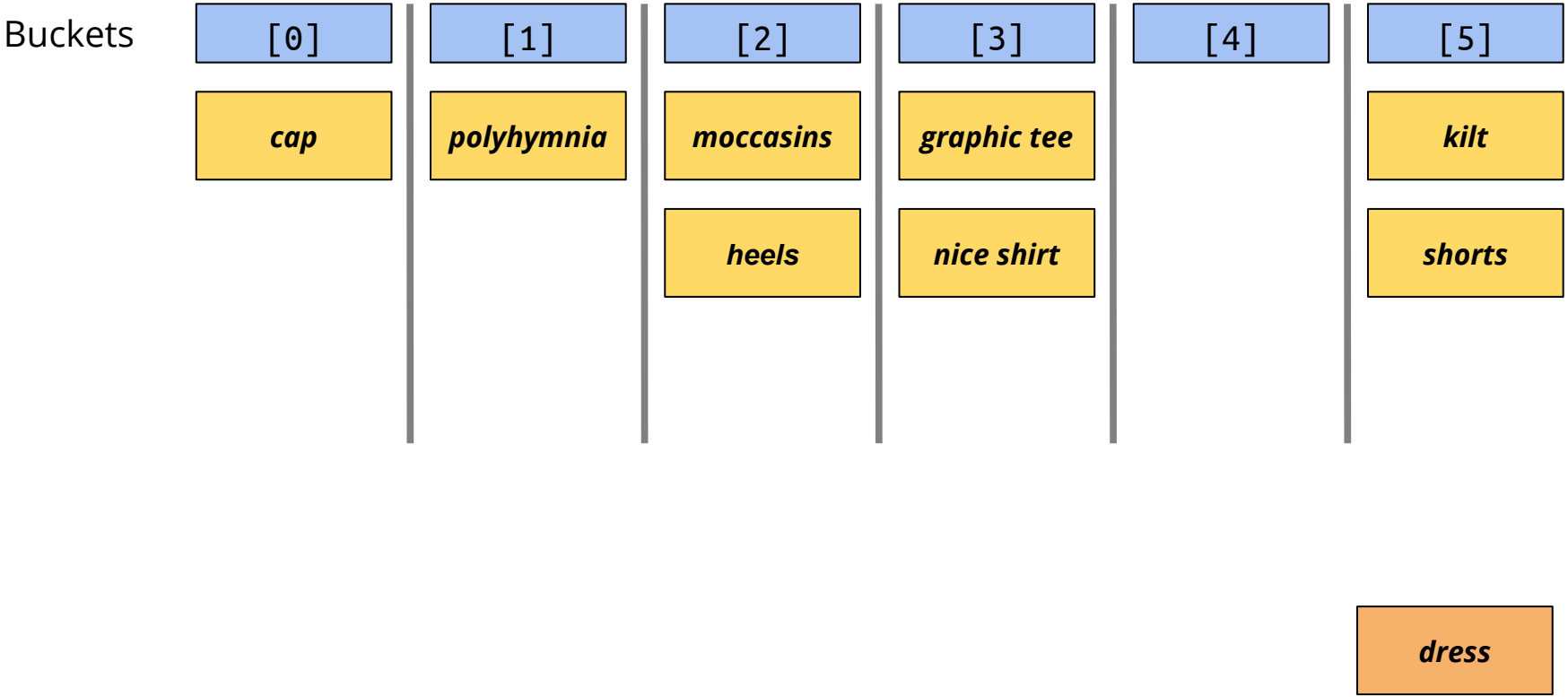
```
int hashCode(string elem)
```

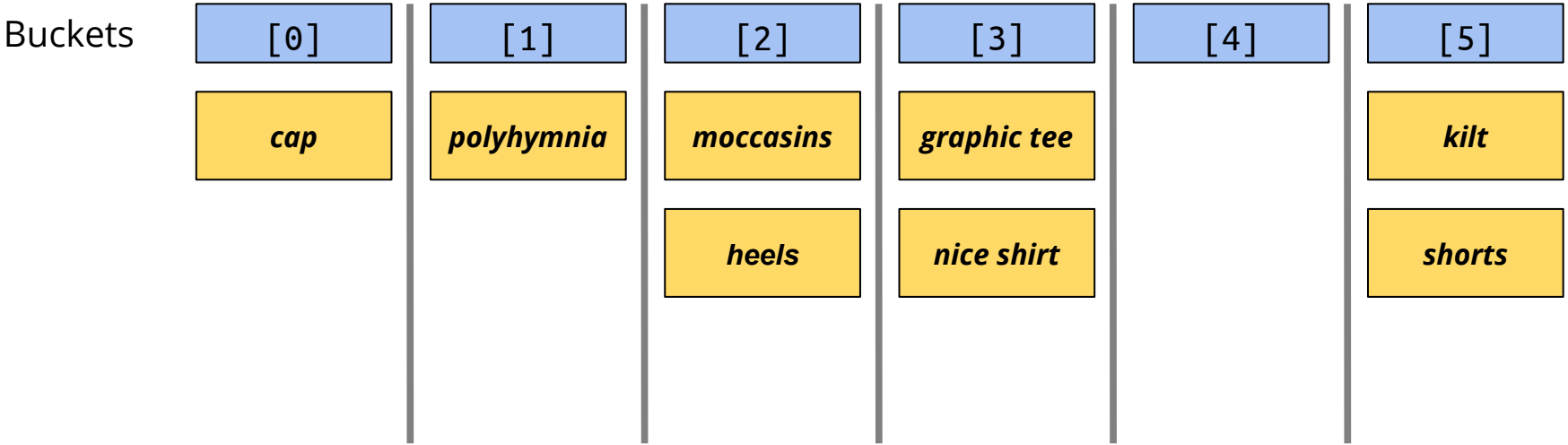



```
bool OurHashSet::contains(const string& value) const {
    int preHash = hashCode(value);
    int hash = preHash % buckets.size();

    for (string elem: buckets[hash]) {
        if (elem == value) return true;
    }

    return false;
}
```

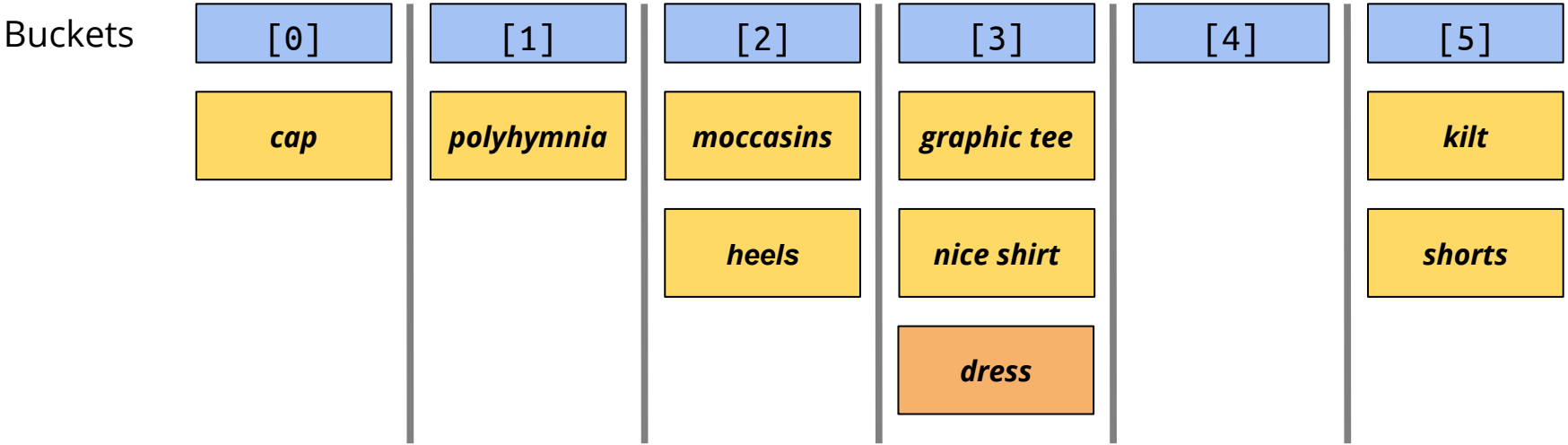

Buckets
[0]
[1]
[2]
[3]
[4]
[5]
cap
polyhymnia
moccasins
graphic tee
kilt
heels
nice shirt
shorts
dress




```
void OurHashSet::add(const string& value) {
    int preHash = hashCode(value);
    int hash = preHash % buckets.size();
    buckets[hash] += value;
}
```

dress

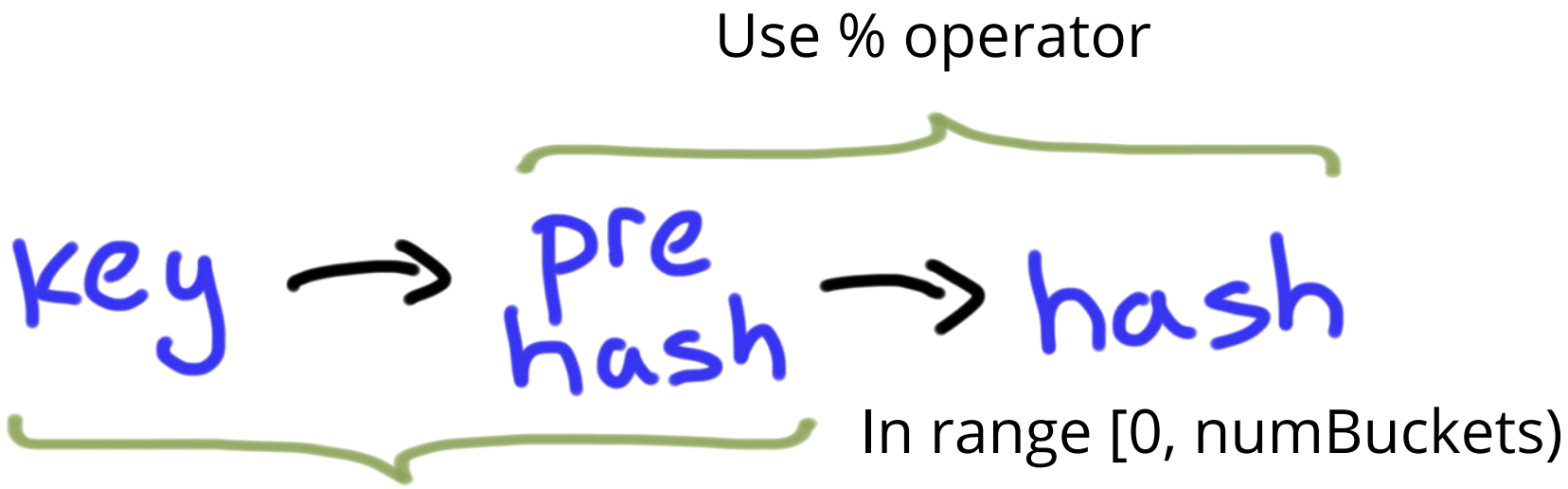
hash: 3




```
void OurHashSet::add(const string& value) {
    int preHash = hashCode(value);
    int hash = preHash % buckets.size();
    buckets[hash] += value;
}
```

The ~~Sorting~~ Hat
Hashing

Enchanted hat that once belonged to Godric Gryffindor and ~~sorts~~ Hashes students into Hogwarts houses


Use % operator
key
pre hash
hash
In range [0, numBuckets)
Generate a really large (positive) number

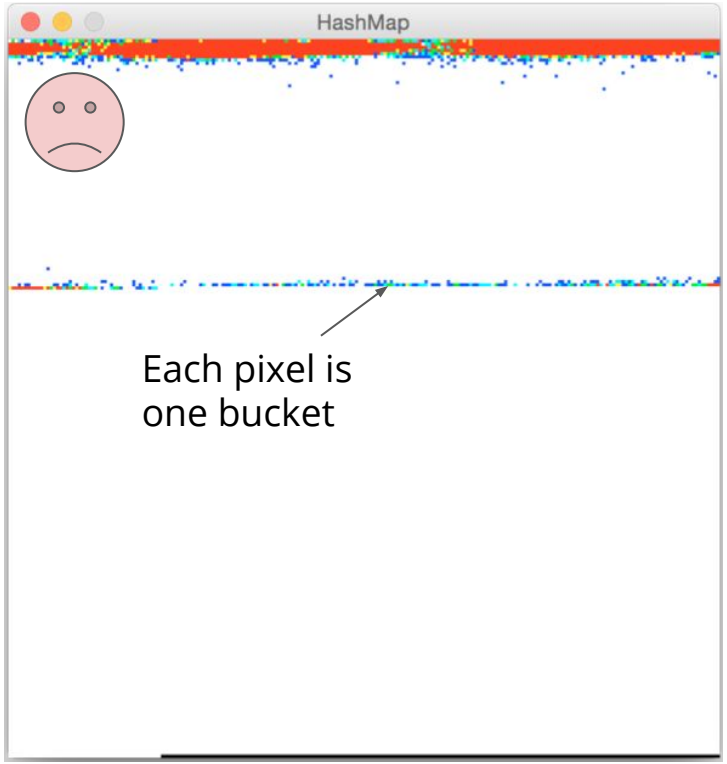
Let's make our own hashing function for `string`!

preHash: sum of all character values!

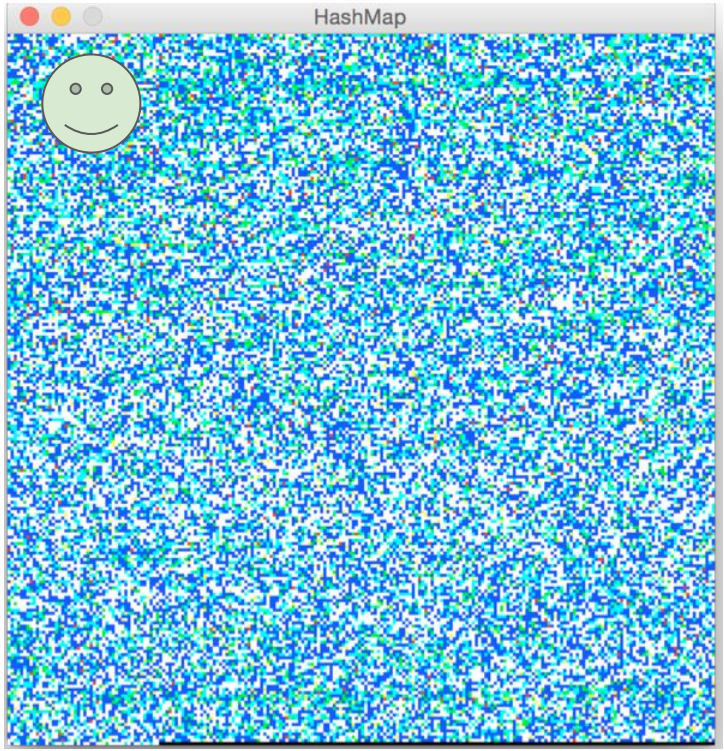
```
prehash =  a[0] + a[1] + ··· + a[n − 1]
return (prehash % numBuckets)
```

Experiment: I hashed 50 thousand Wikipedia article titles into 50 thousand buckets and looked at the number of collisions in each bucket.

preHash: sum of all characters in string



preHash: add each char weighted by 31^i



0 2 4

1 3 >= 5

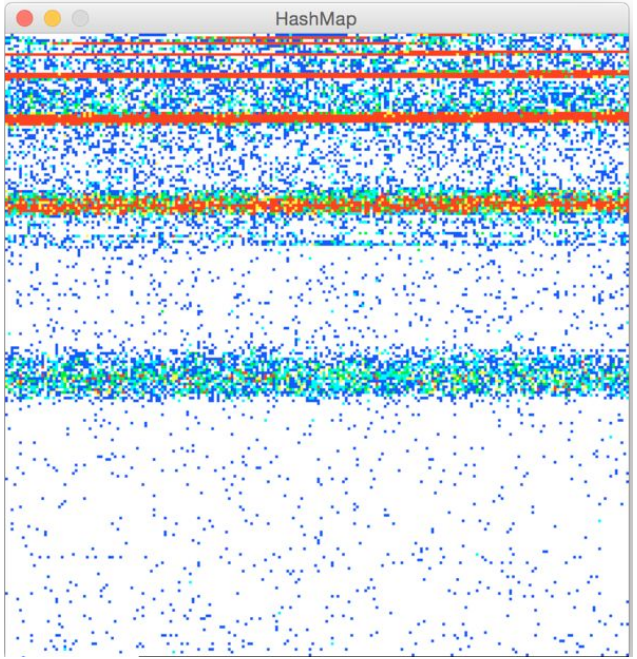
Number of collisions

The clear winner!

```
prehash = a[0] + 31 · a[1] + 31^2 · a[2] + ··· + 31^n · a[n
return (prehash % numBuckets)
```

Why 31? Why not something different?

$$a[0] + 2 \cdot a[1] + 2^2 \cdot a[2] + \cdots + 2^n \cdot a[n]$$



Lesson: Don’t build your own hash function!

What's the Big-O of these functions if our hash function **distributes well** and **`numBuckets` ≥ `numElements`**?

`put()`	`get()`	`remove()`
O(1)	O(1)	O(1)

Set Efficiency

	put()	**get()**	**remove()**
Linked List	O(1)	O(N)	O(N)
BST	O(log(N))	O(log(N))	O(log(N))
Hash	O(1)	O(1)	O(1)

Questions?

Another use of hashing...

Cybersecurity!