Hash Tables

https://cs.pomona.edu/classes/cs140/

Outline

Topics and Learning Objectives

- Discuss hash tables
- Discuss collision handling methods

Exercise

- Collision probabilities

Programming Languages

Python (2 and 3): Built-In (`{}` and `set()`)

[The Google Swiss Table is better.]

C++: `unordered_map` and `unordered_set`

Java: `HashMap` and `HashSet`

Rust: `HashMap` and `HashSet`

Swift: `Dictionary` and `Set`

JavaScript: Built-In hash map `{}` and a set object `Set()`

C#: `Dictionary` and `HashSet`

"To get this out of the way: you should probably just use Vec or HashMap."

-- Rust Documentation

```
hash_table = {}
```

Indices	
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Entries			
i	Hash Value	Key	p

```
hash_table = {}

hash_table["Tony"] = 1
```

Indices	
0	
1	
2	
3	
4	
5	
6	
7	
8	
9	

Entries

i	Hash Value	Key	p

```python
hash_table = {}

hash_table["Tony"] = 1
```

Indices	
0	
1	
2	
3	
4	
5	
6	
7	
8	0
9	

Entries

i	Hash Value	Key	p
0	2513555521146574408	"Tony"	

1

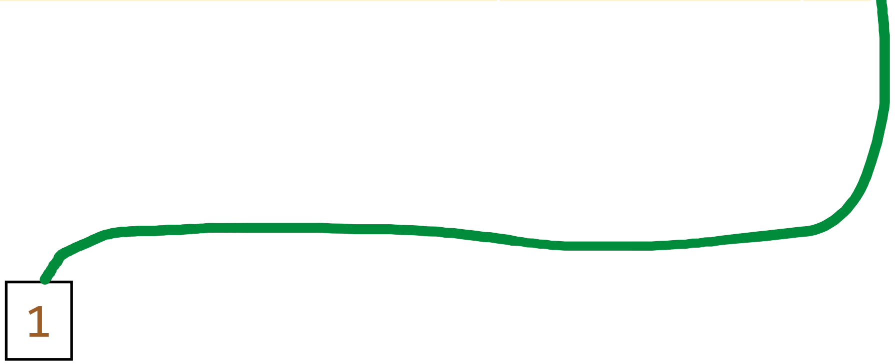
```python
hash_table = {}


hash_table["Tony"] = 1

hash_table["Anthony"] = ("a", "b", "c")
```

Indices

0	
1	
2	
3	
4	
5	1
6	
7	
8	0
9	

Entries

i	Hash Value	Key	p
0	2513555521146574408	"Tony"	

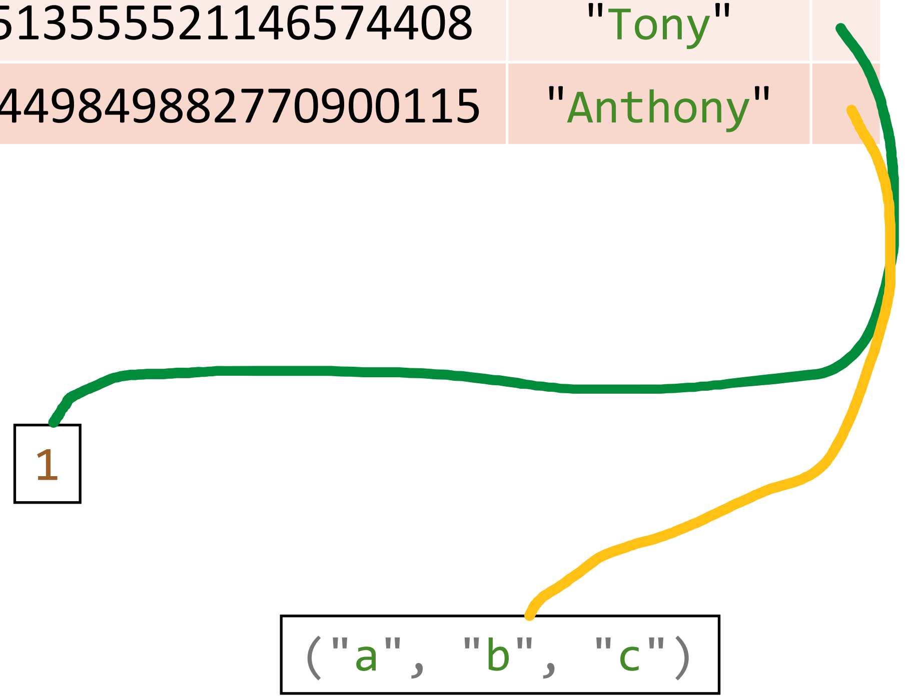


```python
hash_table = {}

hash_table["Tony"] = 1
hash_table["Anthony"] = ("a", "b", "c")
```

Indices	
0	
1	
2	
3	
4	
5	1
6	
7	
8	0
9	

Entries

i	Hash Value	Key	p
0	2513555521146574408	"Tony"	
1	-5449849882770900115	"Anthony"	

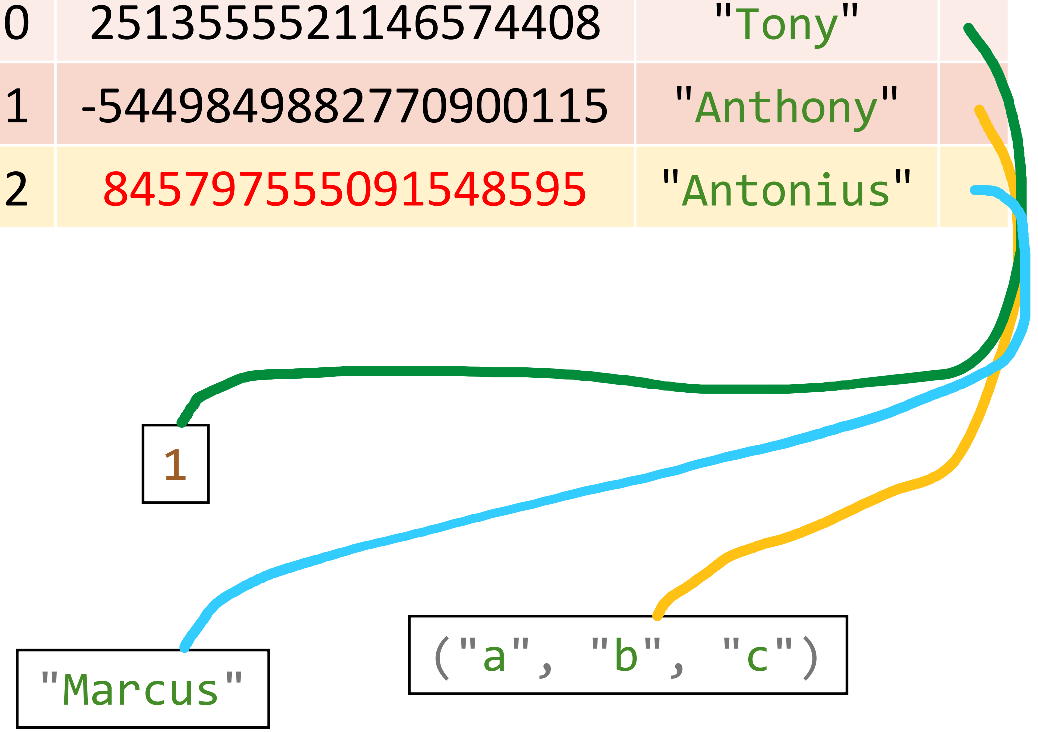


```python
hash_table = {}

hash_table["Tony"] = 1
hash_table["Anthony"] = ("a", "b", "c")
hash_table["Antonius"] = "Marcus"
```

Indices	
0	
1	
2	
3	
4	
5	1
6	
7	
8	0
9	

Entries

i	Hash Value	Key	p
0	2513555521146574408	"Tony"	
1	-5449849882770900115	"Anthony"	
2	845797555091548595	"Antonius"	



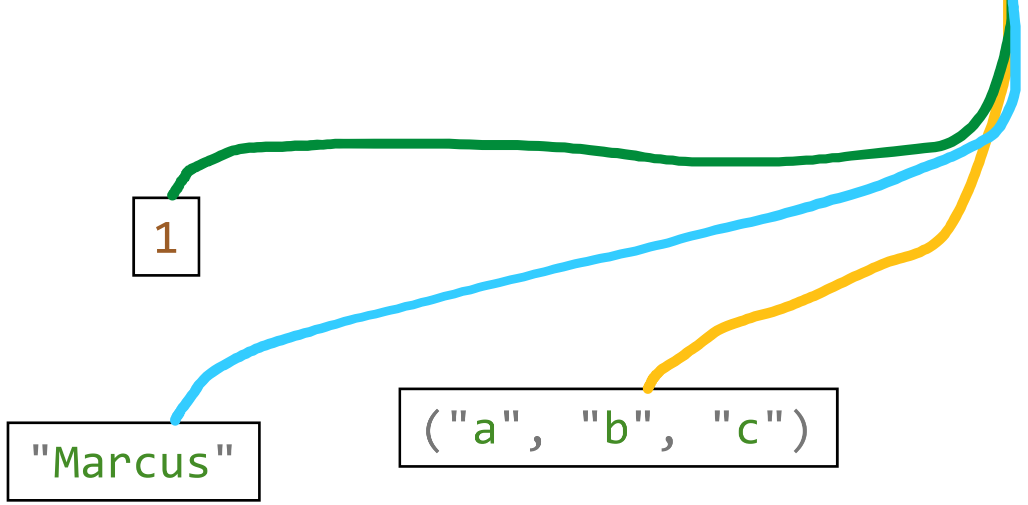
Create a sequence of hash values.

```python
hash_table = {}

hash_table["Tony"] = 1
hash_table["Anthony"] = ("a", "b", "c")
hash_table["Antonius"] = "Marcus"
```

Indices	
0	
1	2
2	
3	
4	
5	1
6	
7	
8	0
9	

Entries

i	Hash Value	Key	p
0	2513555521146574408	"Tony"	
1	-5449849882770900115	"Anthony"	
2	845797555091548595	"Antonius"	

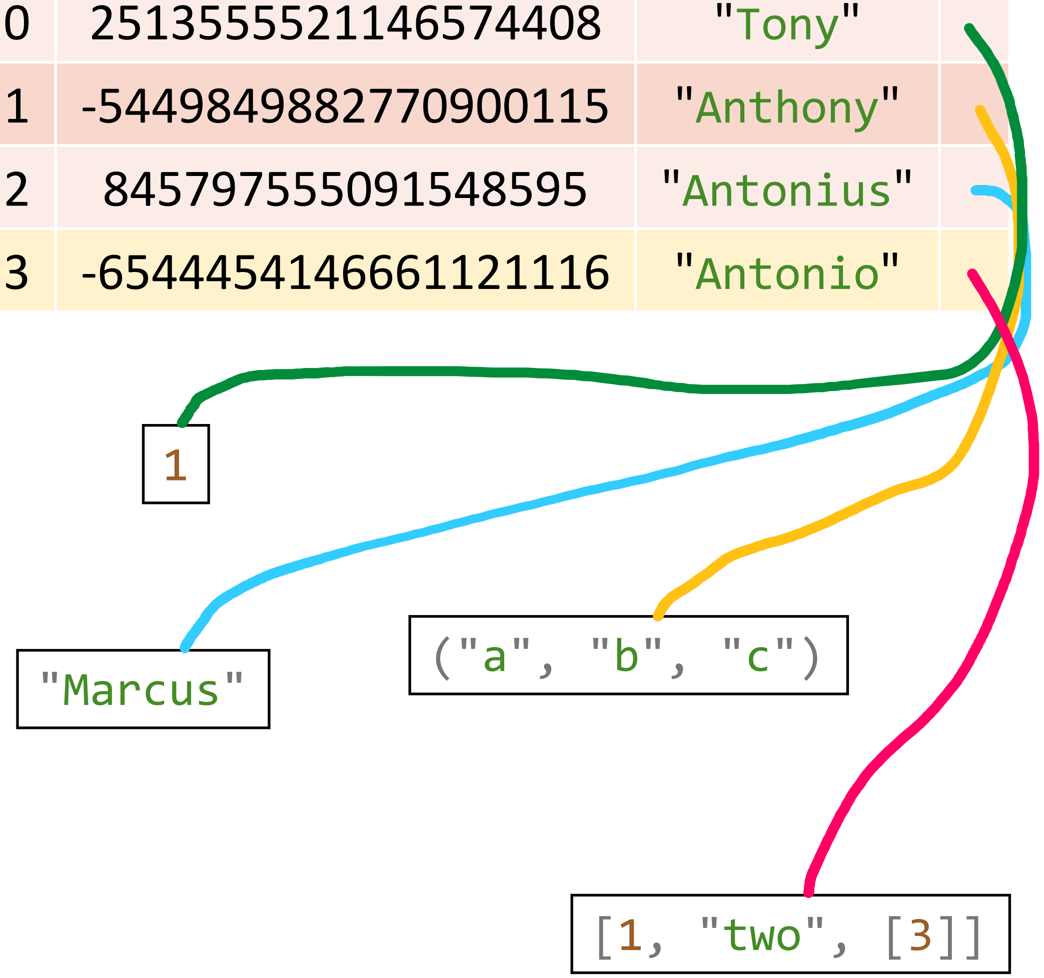


```
hash_table = {}


hash_table["Tony"] = 1
hash_table["Anthony"] = ("a", "b", "c")
hash_table["Antonius"] = "Marcus"
hash_table["Antonio"] = [1, "two", [3]]
```

Indices	
0	
1	2
2	
3	
4	
5	1
6	3
7	
8	0
9	

Entries

i	Hash Value	Key	p
0	2513555521146574408	"Tony"	
1	-5449849882770900115	"Anthony"	
2	845797555091548595	"Antonius"	
3	-6544454146661121116	"Antonio"	

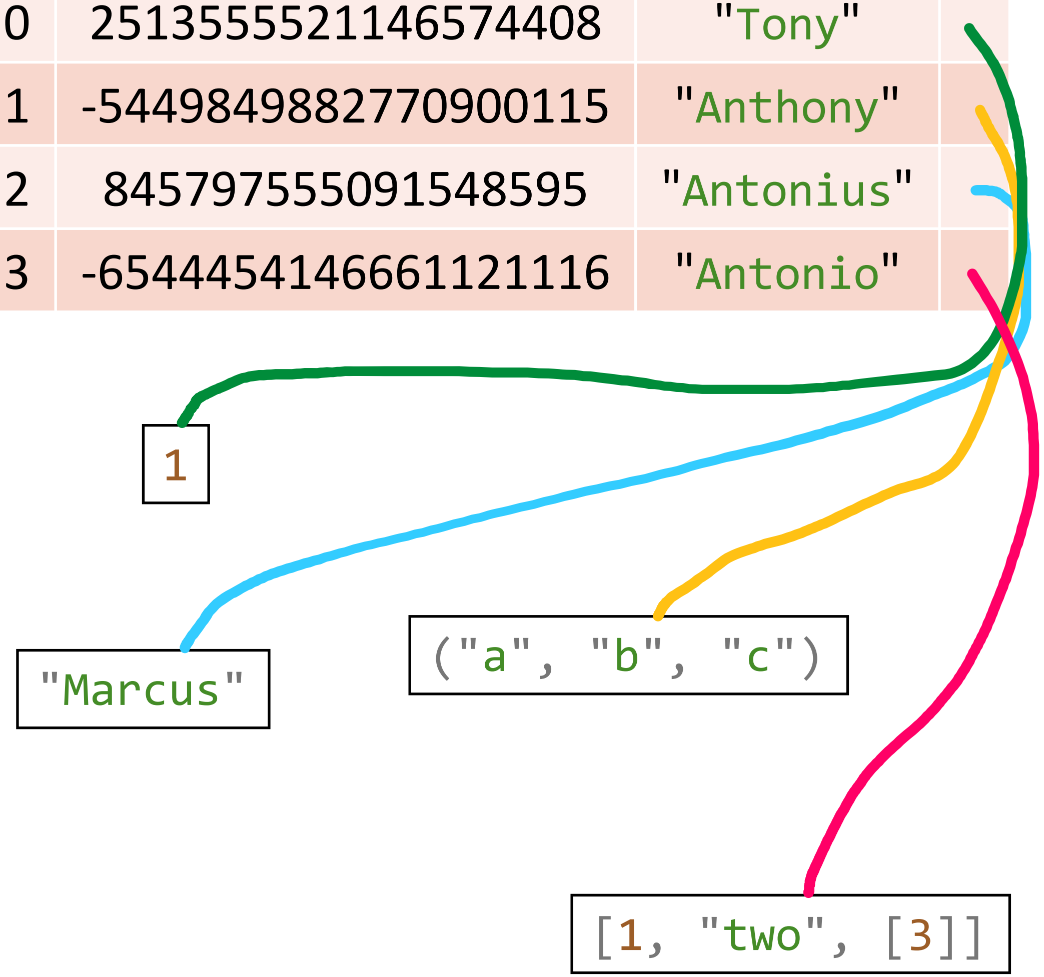


```python
hash_table = {}

hash_table["Tony"] = 1
hash_table["Anthony"] = ("a", "b", "c")
hash_table["Antonius"] = "Marcus"
hash_table["Antonio"] = [1, "two", [3]]
```

Indices	
0	
1	2
2	
3	
4	
5	1
6	3
7	
8	0
9	

Entries

i	Hash Value	Key	p
0	2513555521146574408	"Tony"	
1	-5449849882770900115	"Anthony"	
2	8457975550915485895	"Antonius"	
3	-6544454146661121116	"Antonio"	

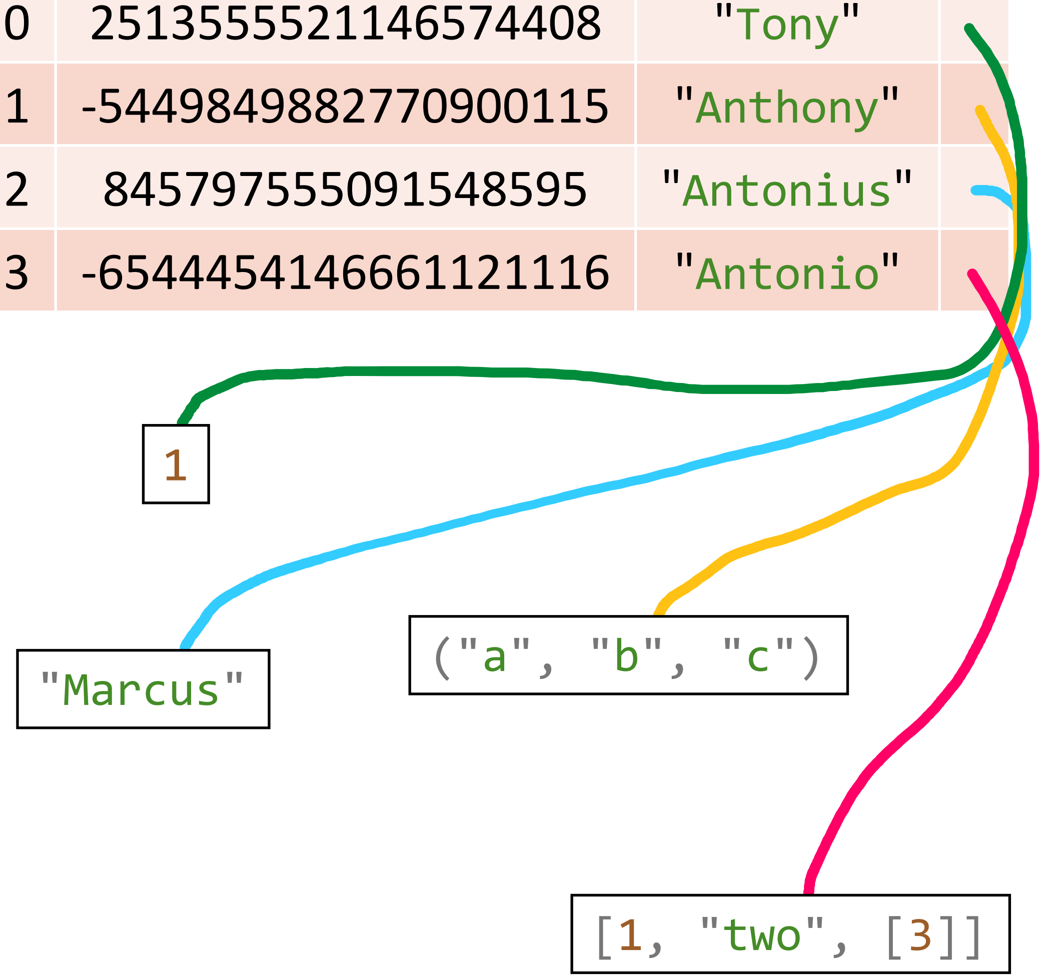


```
hash_table = {}

hash_table["Tony"] = 1
hash_table["Anthony"] = ("a", "b", "c")
hash_table["Antonius"] = "Marcus"
hash_table["Antonio"] = [1, "two", [3]]

# Perform a lookup
get_value = hash_table["Anthony"]
```

Indices	
0	
1	2
2	
3	
4	
5	1
6	3
7	
8	0
9	

Entries

i	Hash Value	Key	p
0	2513555521146574408	"Tony"	
1	-5449849882770900115	"Anthony"	
2	845797555091548595	"Antonius"	
3	-6544454146661121116	"Antonio"	



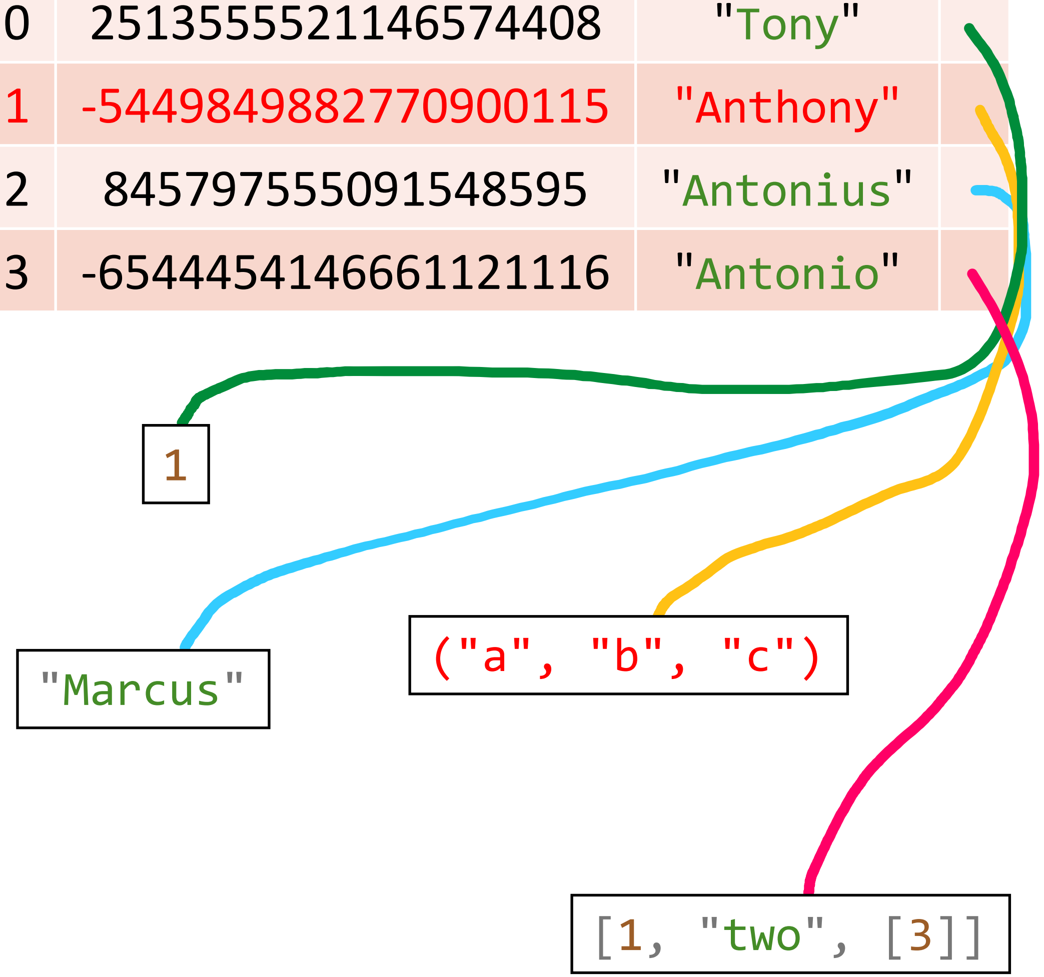
```python
hash_table = {}

hash_table["Tony"] = 1
hash_table["Anthony"] = ("a", "b", "c")
hash_table["Antonius"] = "Marcus"
hash_table["Antonio"] = [1, "two", [3]]

# Perform a lookup
get_value = hash_table["Anthony"]

# Remove an element
del hash_table["Anthony"]
```

Indices

0	
1	2
2	
3	
4	
5	1
6	3
7	
8	0
9	

Entries

i	Hash Value	Key	p
0	2513555521146574408	"Tony"	
1	-5449849882770900115	"Anthony"	
2	8457975550915448595	"Antonius"	
3	-6544454146661121116	"Antonio"	

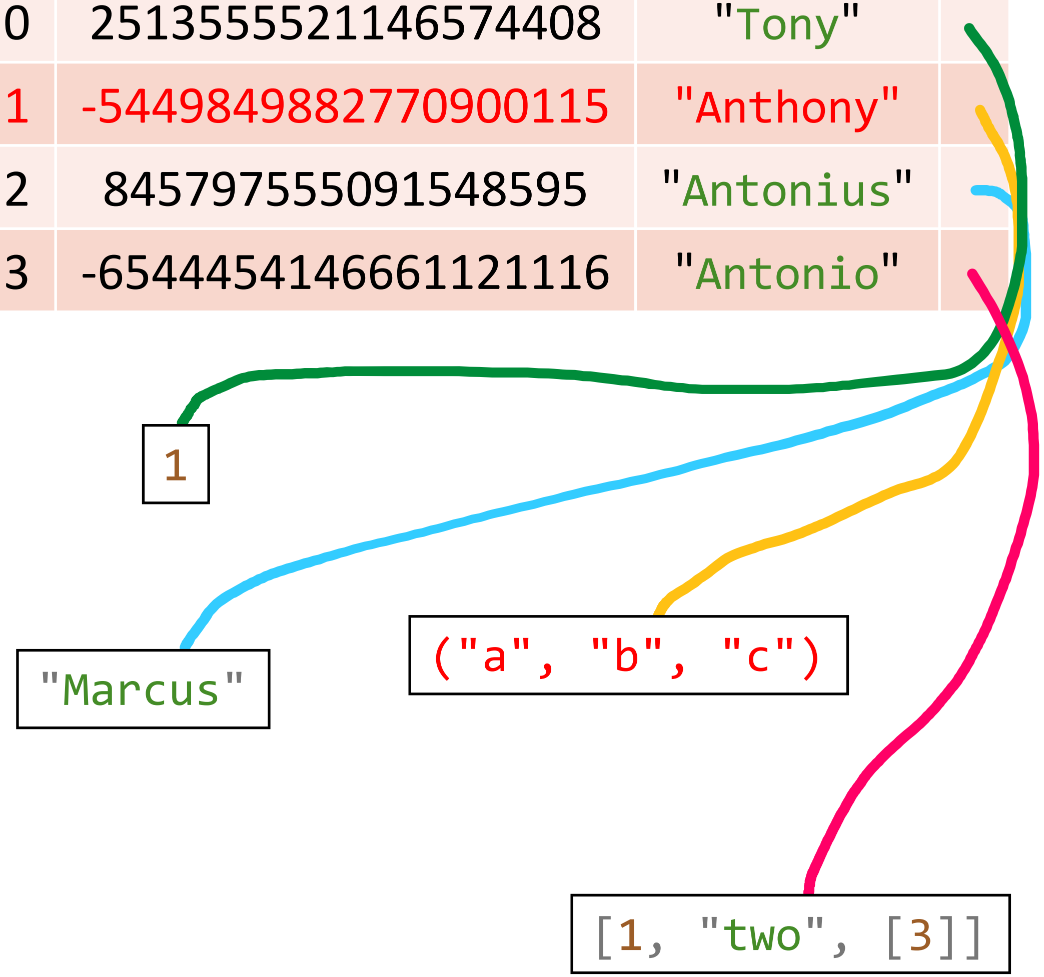


```python
hash_table = {}

hash_table["Tony"] = 1
hash_table["Anthony"] = ("a", "b", "c")
hash_table["Antonius"] = "Marcus"
hash_table["Antonio"] = [1, "two", [3]]

# Perform a lookup
get_value = hash_table["Anthony"]

# Remove an element
del hash_table["Anthony"]

get_value2 = hash_table["Antonius"]
```

Indices	
0	
1	2
2	
3	
4	
5	
6	3
7	
8	0
9	

Entries

i	Hash Value	Key	p
0	2513555521146574408	"Tony"	
1	-5449849882770900115	"Anthony"	
2	845797555091548595	"Antonius"	
3	-6544454146661121116	"Antonio"	

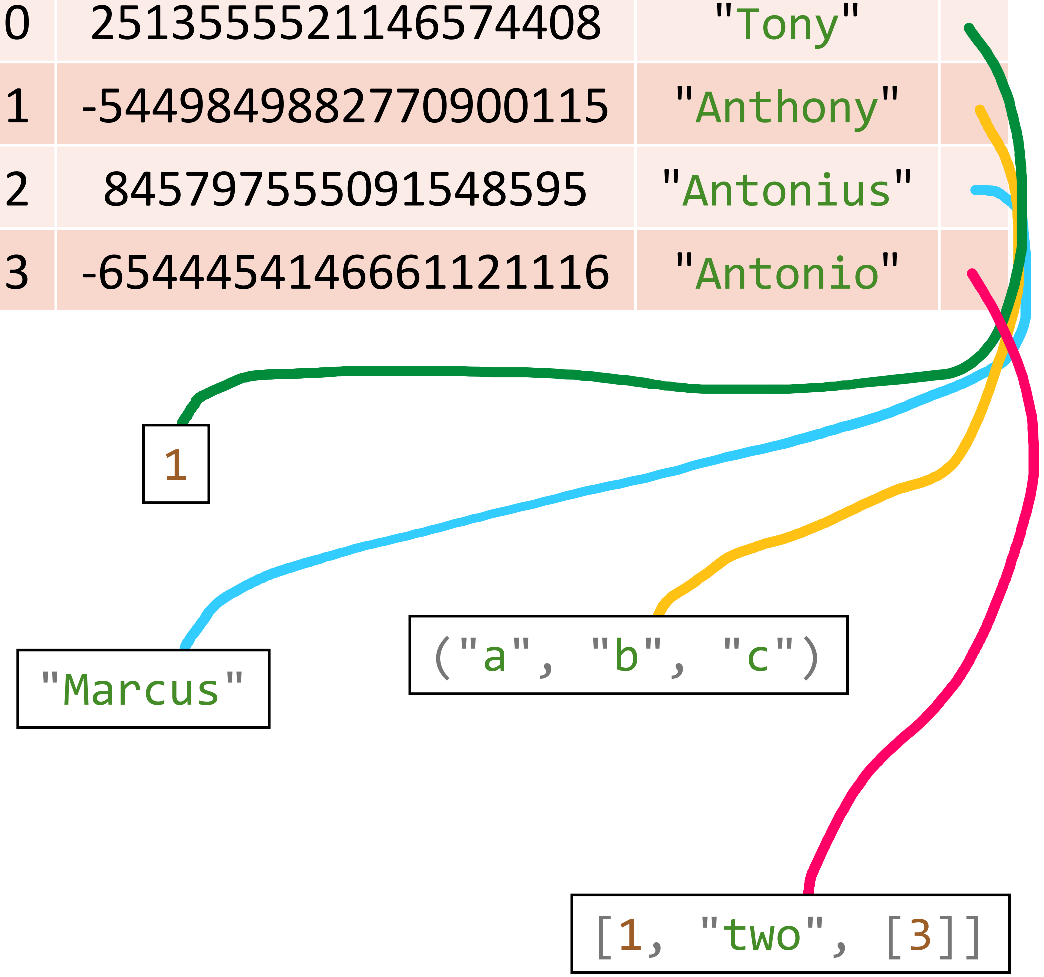


```python
hash_table = {}

hash_table["Tony"] = 1
hash_table["Anthony"] = ("a", "b", "c")
hash_table["Antonius"] = "Marcus"
hash_table["Antonio"] = [1, "two", [3]]

some_list = [1, "two", (1, 1, 1)]
hash_table[some_list] = "Antonio"
some_list.append("hi class")
get_value = hash_table[some_list]
```

Indices	
0	
1	2
2	
3	
4	
5	1
6	3
7	
8	0
9	

Entries

i	Hash Value	Key	p
0	2513555521146574408	"Tony"	
1	-5449849882770900115	"Anthony"	
2	845797555091548595	"Antonius"	
3	-6544454146661121116	"Antonio"	



Should this work? What would it do?

https://github.com/python/cpython/blob/master/Objects/dictobject.c

Common Hash Function

```
def djb2(s):
    hash = 5381 # some prime number
    magic = 33  # magic number that works well
    for c in s:
        hash = hash * magic + ord(c)
    return hash & 0xFFFFFFFF
```

https://softwareengineering.stackexchange.com/questions/49550/which-hashing-algorithm-is-best-for-uniqueness-and-speed

Hash Tables

- One of the most useful and used data structures
- They do **not** support many operations
- But they are amazing at the operations they do support

- They act like an array with a couple of key differences

Hash Tables

Operations:

- Insert
- Delete
- Look-up

O(1)

What are they not good for?

Guaranteed constant running time for those operations if:

1. If the hash table is properly implemented, and
2. The data is non-pathological.

Example Applications

Removing Duplicates

- Given a stream of objects
- Don’t add object if it already exists

- Distinct visitors to a web site
- Blacklist or whitelist
- Creating an efficient web crawler

Two-Sum Problem

- Given an array of integers A and a target sum T
- Goal: determine if any two numbers sum to T
- What is a naïve approach to this problem?
- What is a slightly better approach?
- What is an optimal approach based on hash-tables?

Other applications

- Used for symbol tables in compilers
- In search algorithms you can ensure that you don't test the same configuration twice

Great Data Structure—Easy to butcher

- Let U be the universe of all possible objects
  - (all possible IP address, all possible student names, all chess configurations, etc.)

- We want to maintain an evolving subset S of U that is a *reasonable* size

- Naïve solution #1: is to create an array that has equal to |U|
  - No collisions but requires a huge amount of space
- Naïve solution #2: use a linked list instead
  - Relatively memory efficient, but everything collides

Great Data Structure—Easy to butcher

- Let U be the universe of all possible objects
  - (all possible IP address, all possible student names, all chess configurations, etc.)

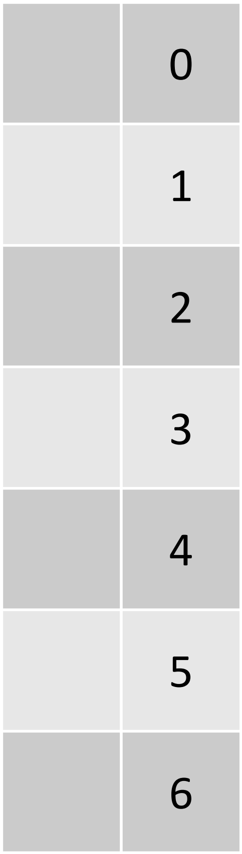
- We want to maintain an evolving subset S of U that is a *reasonable* size

Hash table:

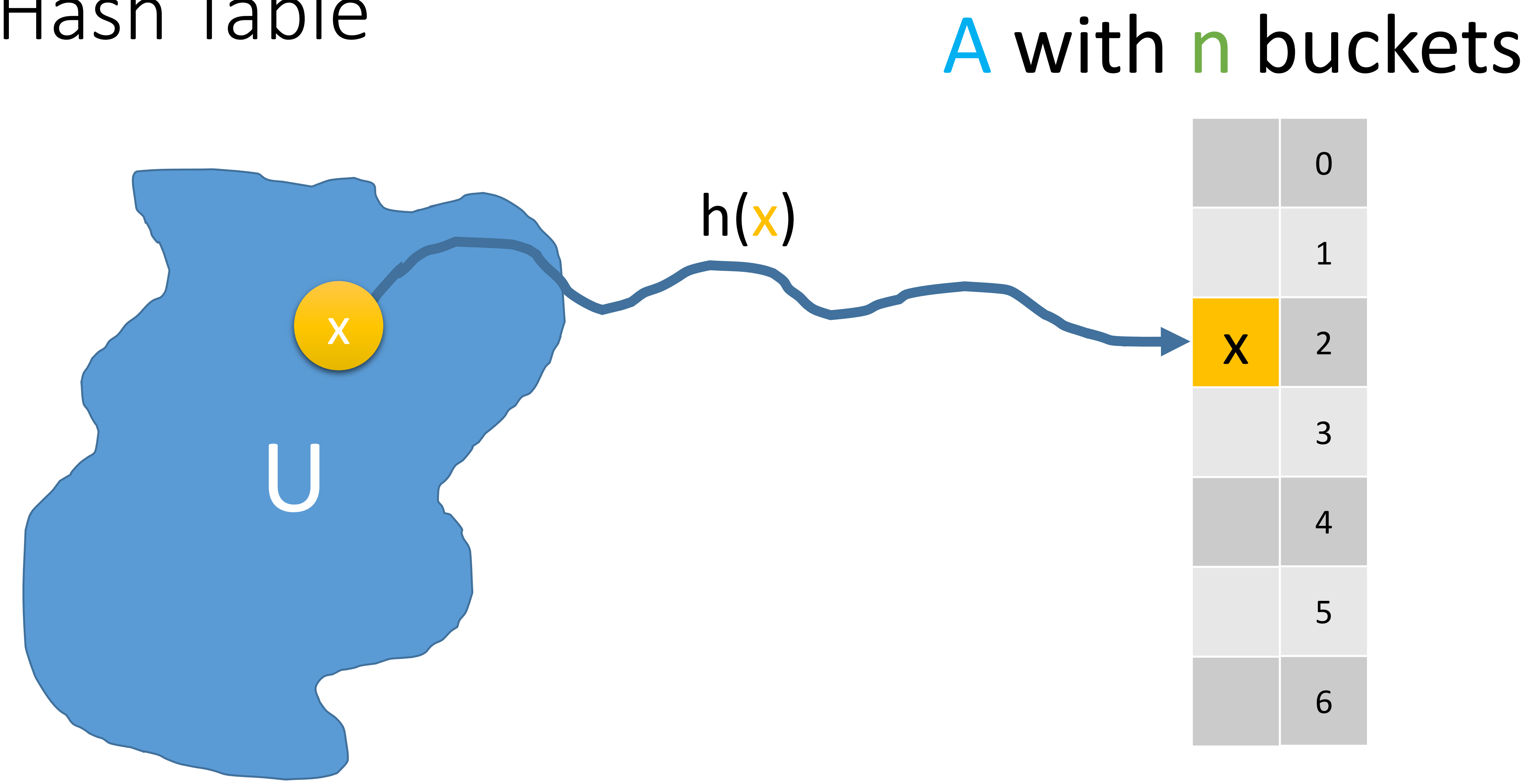
- Let n be approximately equal to |S|, where n is the # of buckets
- Choose a hash function `h(x) → {0, 1, .., n-1}` where x is an object in U
- Use an array A of length n, and store objects at A[h(x)]

Hash Table

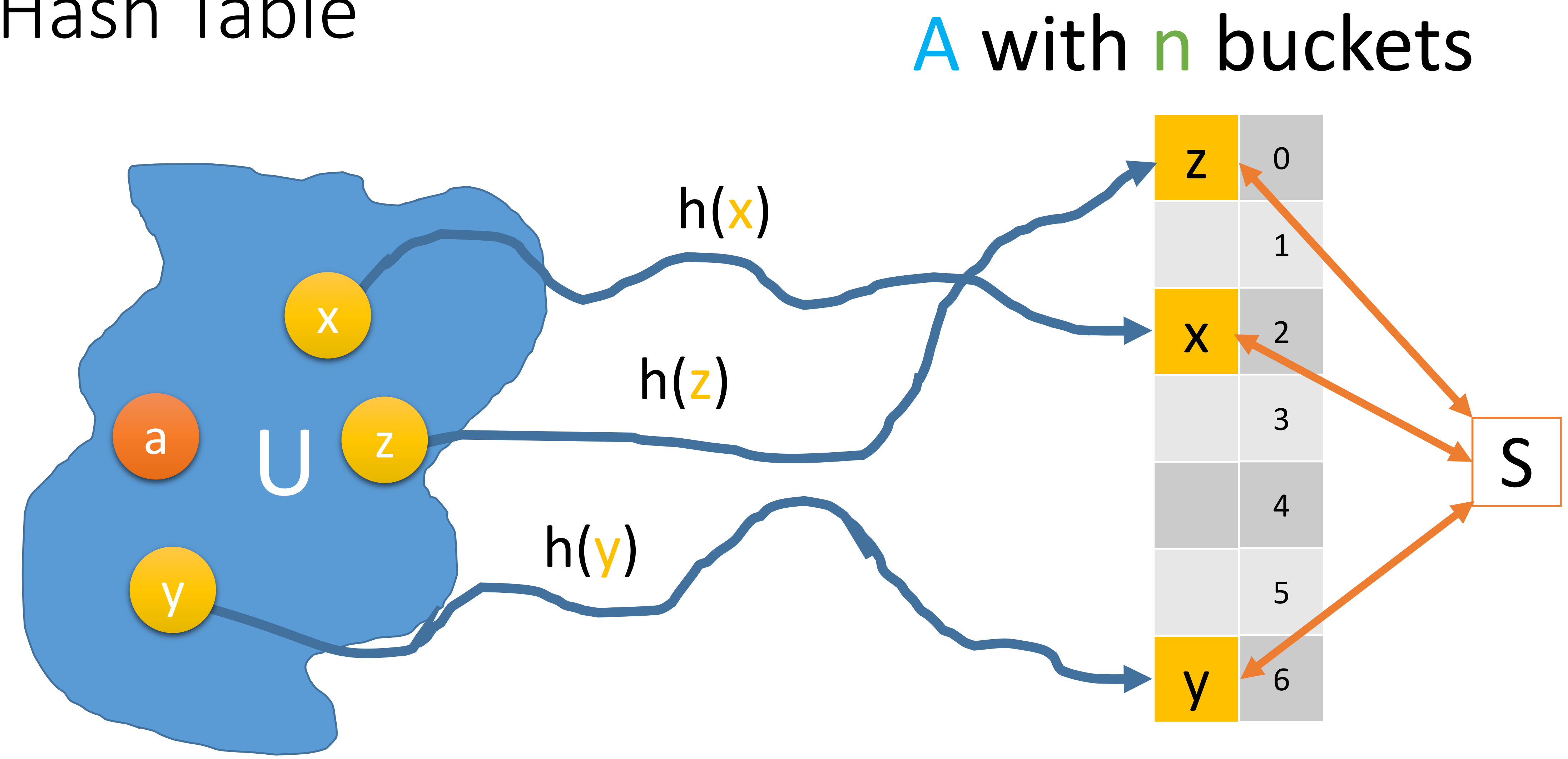
A with n buckets



Hash Table



Hash Table



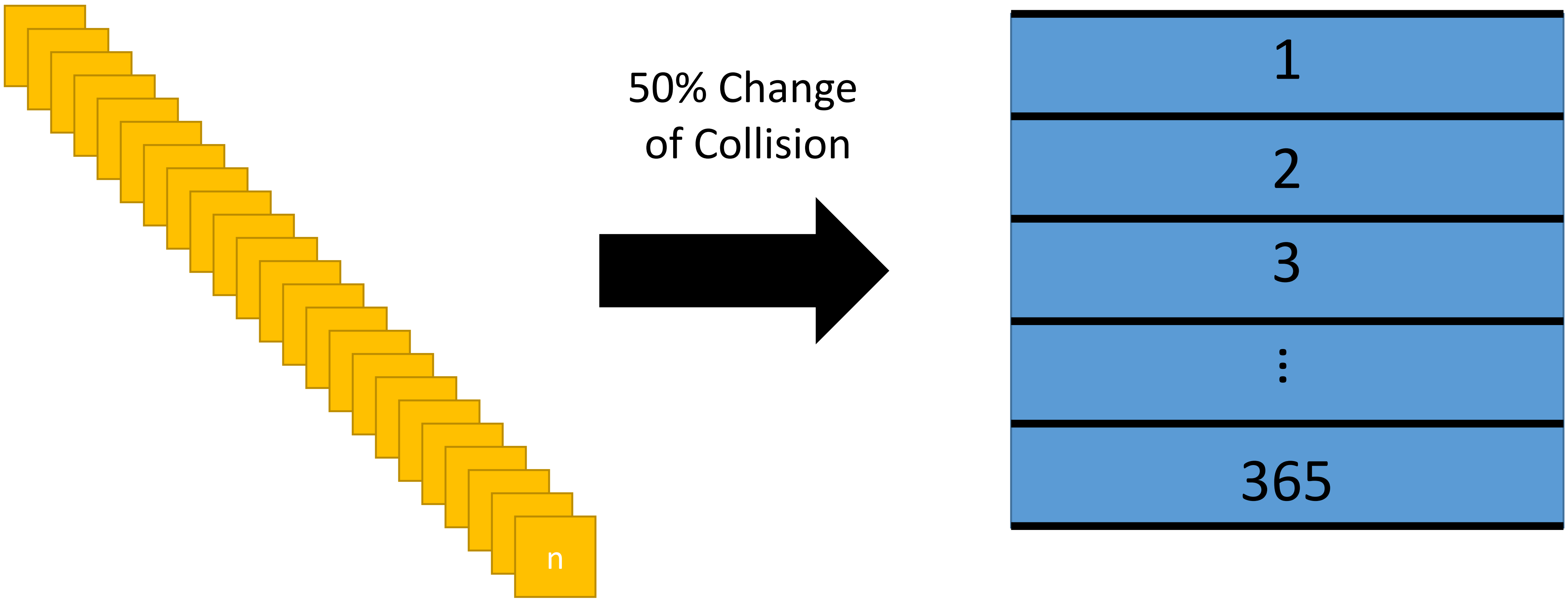
Collisions

- What if two keys (objects) result in the same index?
- Is this really a problem? Does it happen very often?

Birthday problem

- Consider n people with birthdays distributed uniformly at random.
- How large does n need to be before there is at least a 50% chance that two people have the same birthday?

Birthday Problem



Exercise Question 1

- What if two keys (objects) result in the same index?
- Is this really a problem? Does it happen very often?

Birthday problem

- Consider n people with birthdays distributed uniformly at random.
- How large does n need to be before there is at least a 50% chance that two people have the same birthday?

Exercise Question 1

- What if two keys (objects) result in the same index?
- Is this really a problem? Does it happen very often?

Birthday problem

- Consider n people with birthdays distributed uniformly at random.
- How large does n need to be before there is at least a 50% chance that two people have the same birthday?
  a. 367
  b. 57
  c. 184
  d. 23

Break Video

Exercise Question 1

- What if two keys (objects) result in the same index?
- Is this really a problem?

Birthday problem

- Consider n people with birthdays distributed uniformly at random.
- How large does n need to be before there is at least a 50% chance that two people have the same birthday?
  a. 367 → 100%
  b. 57 → 99%
  c. 184 → 99.9999%
  d. 23 → 50%

Exercise Question 2

$$\prod_{i=1}^{x} \frac{n-i}{n} \sim e^{-x(x-1)/2n}$$

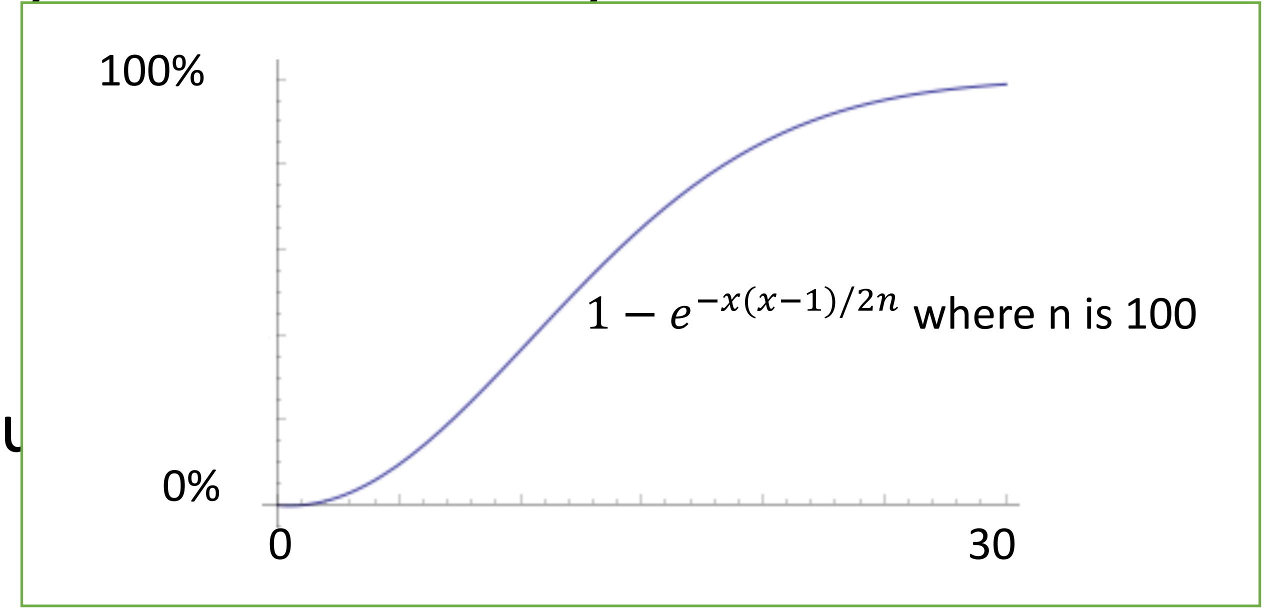
- We have a hash table implemented using an array with 100 buckets.
- Assume that we have a perfect hash function (generates hash values uniformly at random).
- What is the probability of any collisions if we try to store:
  - 10 objects?
  - 20 objects?
  - 30 objects?

- Let’s use a slightly more accurate equation.

Exercise Question 2

$$\prod_{i=1}^{x} \frac{n-i}{n} \sim e^{-x(x-1)/2n}$$

- We have a hash table implemented using an array with 100 buckets.
- Assume that we have a perfect hash function (generates hash values uniformly at random).
- What is the probability of any collisions if we try to store:
  - 10 objects? 36%
  - 20 objects? 85%
  - 30 objects? 99%



- Let's use a slightly more accu

Collisions

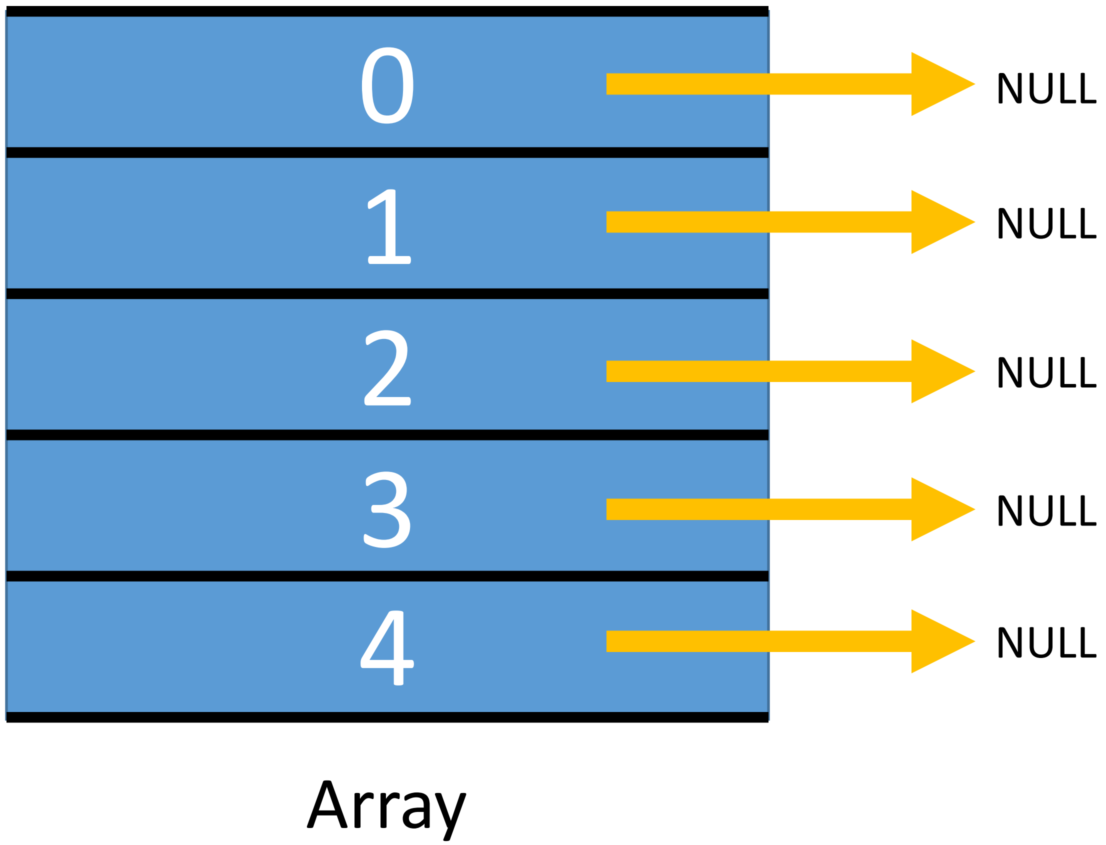
- Even with a uniformly random hash function you still get quite a few collisions with a small data set.
- Collisions occur often, so we need to handle them carefully.

Two common methods for resolving collisions

1. Separate Chaining
2. Open Addressing

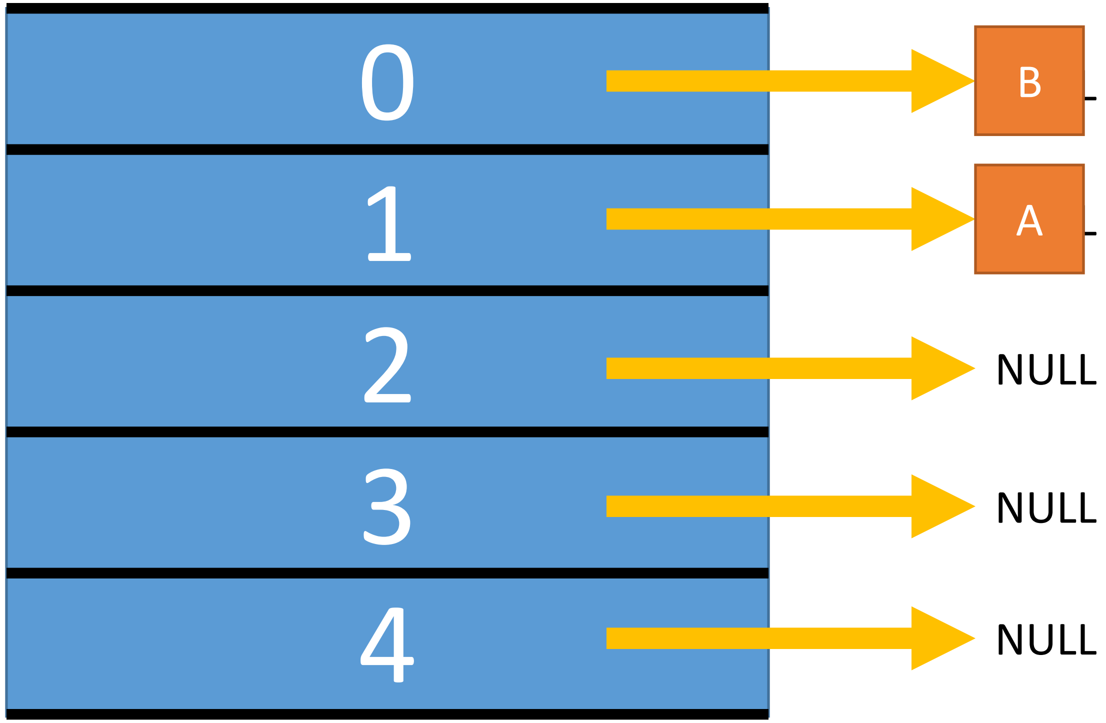
In practice, we use something similar to these (e.g., the Python example)

Method 1: Separate Chaining



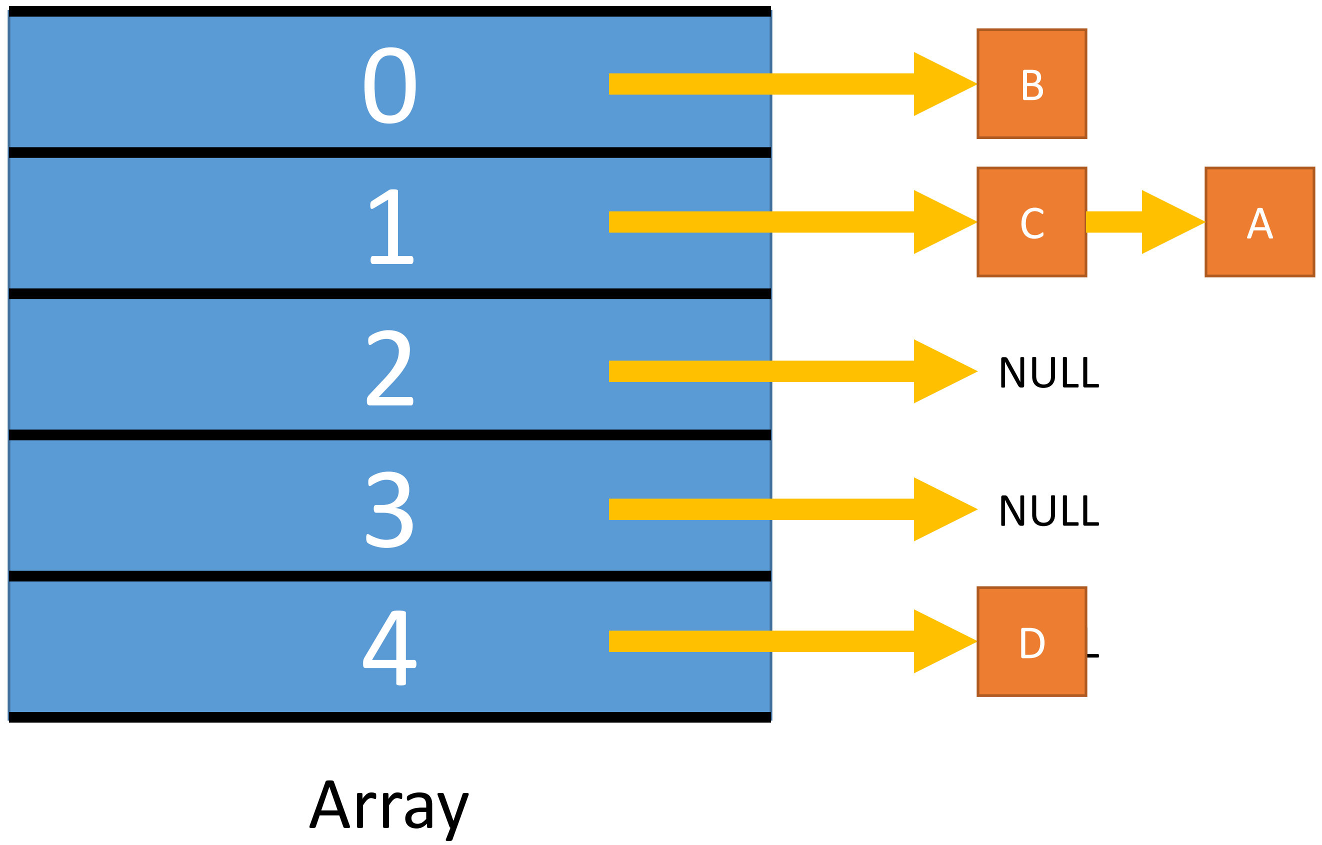
Method 1: Separate Chaining

```
h(A) = 53 726
h(A) % 5 = 1

h(B) = 224 930
h(B) % 5 = 0

h(C) = 23 321
h(C) % 5 = 1
```



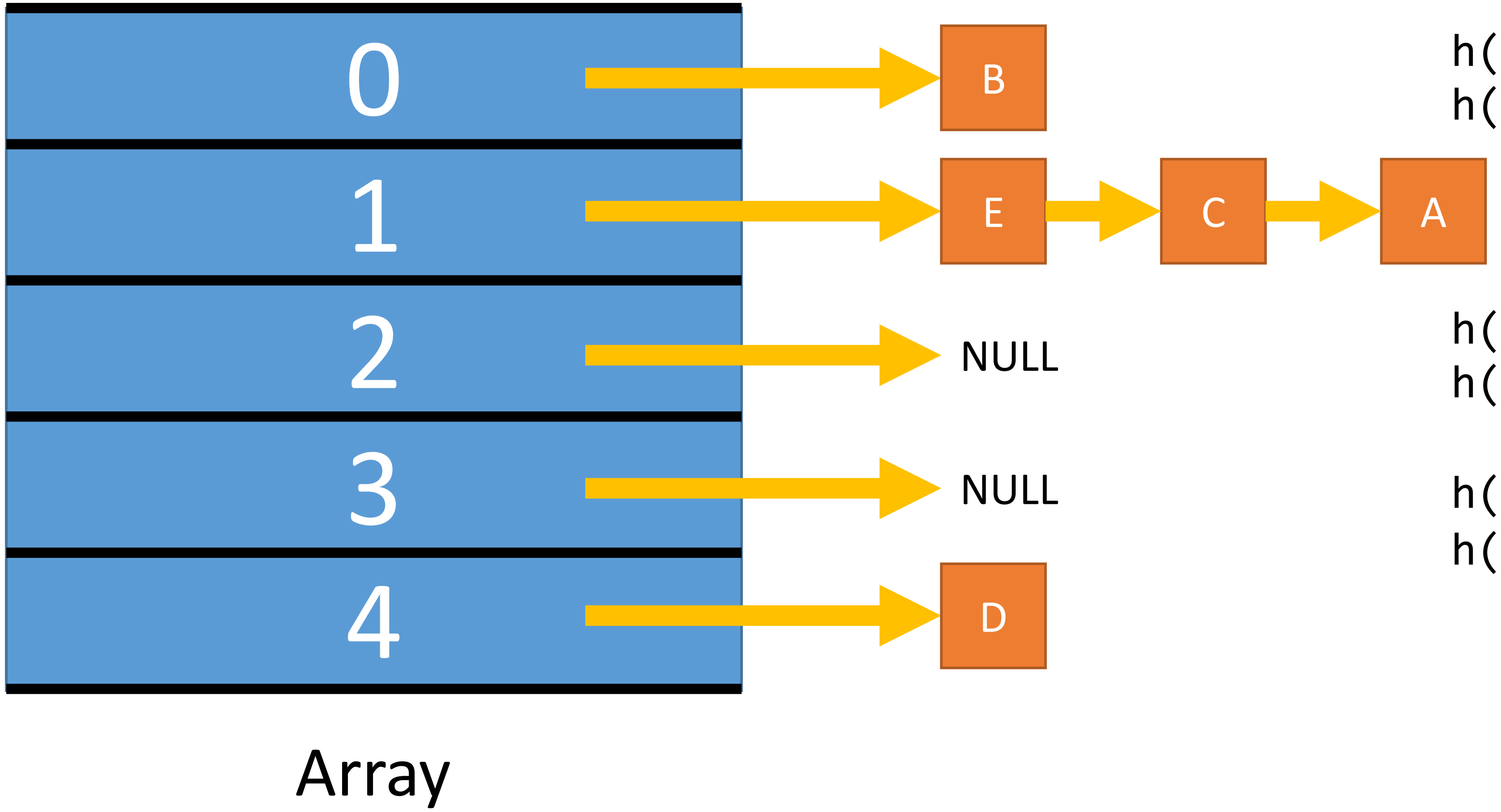
Array

Method 1: Separate Chaining




```
h(A) = 53 726
h(A) % 5 = 1

h(B) = 224 930
h(B) % 5 = 0

h(C) = 23 321
h(C) % 5 = 1

h(D) = 7 894
h(D) % 5 = 4

h(E) = 919 271
h(E) % 5 = 1
```

Method 1: Separate Chaining




```
h(A) = 53 726
h(A) % 5 = 1

h(B) = 224 930
h(B) % 5 = 0

h(C) = 23 321
h(C) % 5 = 1

h(D) = 7 894
h(D) % 5 = 4

h(E) = 919 271
h(E) % 5 = 1
```

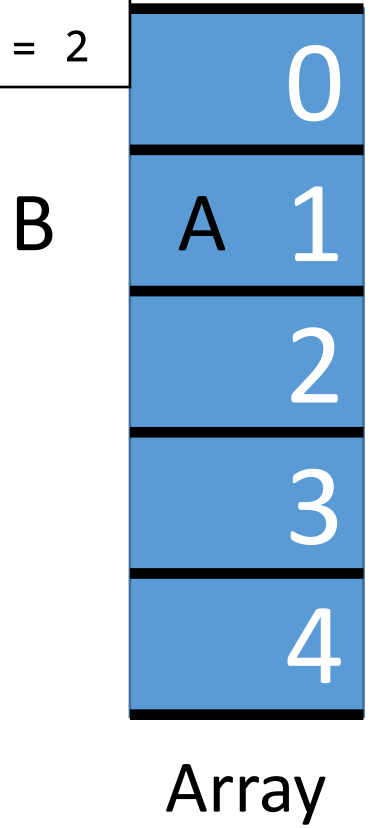
Method 2: Open Addressing

```
h(A) = 53 726
h(A) % 5 = 1

h(B) = 224 936
h(B) % 5 = 1
(h(B) + 1) % 5 = 2
```

Only one object per bucket

Hash functions now specify a sequence

1. Linear probing: call the hash function and find the next available spot in the array



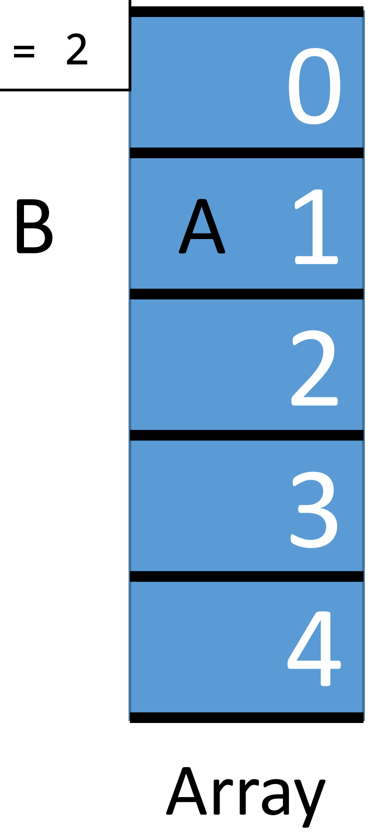
Method 2: Open Addressing

```
h(A) = 53 726
h(A) % 5 = 1

h(B) = 224 936
h(B) % 5 = 1
(h(B) + 1) % 5 = 2
```

Only one object per bucket

Hash functions now specify a sequence

1. Linear probing: call the hash function and find the next available spot in the array

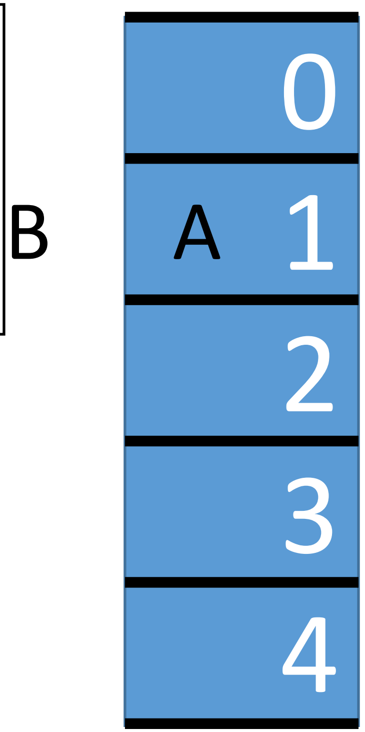


Method 2: Open Addressing

Only one object per bucket

Hash functions now specify a sequence

```
h1(A) = 53 726
h1(A) % 5 = 1

h1(B) = 224 936
h1(B) % 5 = 1
(h1(B) + h2(B)) % 5 = 4
```



Array

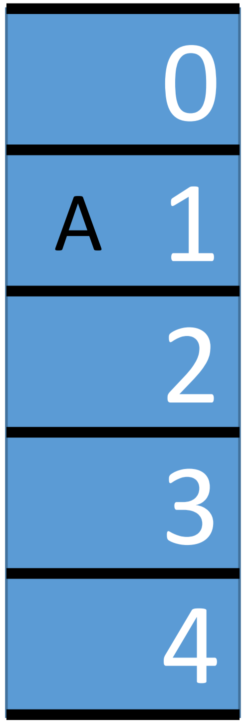
1. Linear probing: call the hash function and find the next available spot in the array
2. Double hashing:
   - Requires **two** hash functions
   - Call the first hash function to get an index
   - Call the second hash function on collision to get an offset
   - Add the offset to the index
   - If there is another collision add the offset again

Method 2: Open Addressing

Only one object per bucket

Hash functions now specify a sequence … B

```
h1(A) = 53 726
h1(A) % 5 = 1

h1(B) = 224 936
h1(B) % 5 = 1
(h1(B) + h2(B)) % 5 = 4
```



Array

1. Linear probing: call the hash function and find the next available spot in the array
2. Double hashing:
   - Requires **two** hash functions
   - Call the first hash function to get an index
   - Call the second hash function on collision to get an offset
   - Add the offset to the index
   - If there is another collision add the offset again

Python 3.6 Dictionaries Advantages

https://stackoverflow.com/questions/39980323/are-dictionaries-ordered-in-python-3-6

```
d = {'timmy': 'red', 'barry': 'green', 'guido': 'blue'}

# Old version
entries = [
        ['--', '--', '--'],
        [-8522787127447073495, 'barry', 'green'],
        ['--', '--', '--'],
        ['--', '--', '--'],
        ['--', '--', '--'],
        [-9092791511155847987, 'timmy', 'red'],
        ['--', '--', '--'],
        [-6480567542315338377, 'guido', 'blue']
]
```

Python 3.6 Dictionaries Advantages

https://stackoverflow.com/questions/39980323/are-dictionaries-ordered-in-python-3-6

```
d = {'timmy': 'red', 'barry': 'green', 'guido': 'blue'}

# Old version
entries = [
        ['--', '--', '--’],
        [-8522787127447073495, 'barry', 'green’],
        ['--', '--', '--’],
        ['--', '--', '--’],
        ['--', '--', '--’],
        [-9092791511155847987, 'timmy', 'red’],
        ['--', '--', '--’],
        [-6480567542315338377, 'guido',
]
```

```
# New version
indices = [None, 1, None, None, None, 0, None, 2]
entries = [
        [-9092791511155847987, 'timmy', 'red’],
        [-8522787127447073495, 'barry', 'green’],
        [-6480567542315338377, 'guido', 'blue’]
]
```

Python 3.6 Dictionaries Advantages

- Uses 30% to 95% less memory

- Resizing the hash table only changes the location of the indices, the indices themselves do not change

- Better cache utilization

Hash Functions

- What makes a good hash function?

- Properties of a good hash function
  1. Should lead to the smallest number of collisions as possible
  2. It shouldn't be too much work to compute the hash
     (required for every lookup, insertion, or deletion)

- What is the worst case for a hash function?

```
def hash_fcn(obj):
    return 1
```

Example

Keys are 10-digit phone numbers, |U| = 10^10, we select n = 10^3

- Terrible hash function: h(x) = 1st 3 digits of x
- OK (not great) hash function: h(x) = x % 1000 (final 3 digits of x)

417-836-6646	417-836-8745
417-836-5438	417-836-4834
417-836-4944	417-836-5789
417-836-5930	417-836-5224
417-836-5026	417-836-4157

Passable Hash Function

Take strings as objects for example

- Hash code could be to create a unique number from the characters
- Compression function would be to take the integer mod n

How do you choose n (the number of buckets)?

- Choose n to be a prime number (on the order of the # of objects to store)
- Don’t choose a value too near to a power of 2
- Don’t choose a value too near to a power of 10

Hash Table Summary

1. Make a big array

2. Create a function that converts elements into integers (hashing)

3. Store elements in the array at the index specified by the hash function

4. Do something interesting if two elements get the same index (collision)

5. Rehash (resize):
   - when the load factor exceeds 75% (rule of thumb)
   - by increasing size by a factor of 1.5 (rule of thumb)