

# FAST LOOKUP WITH HASHTABLES

---

Problem Solving with Computers-II

C++

```
#include <iostream>
using namespace std;
int main(){
    cout<<"Hola Facebook\n";
    return 0;
}
```

# Hastable — Dictionary like data structure

**std::unordered\_set<T>** – stores unique keys, no duplicates

```
std::unordered_set<string> countries = {"USA", "France", "India"};
s.insert("Germany");
s.erase("France");
auto found = find("USA");
```

**std::unordered\_map<Key, Value>** – stores key-value pairs (like a dictionary)

```
unordered_map<string, string> capitals= {{"USA", "Washington"}, {"France", "Paris"}};

capitals["Germany"] = "Berlin"; // insert
capitals.erase("France"); // delete
auto found = find("USA"); // find
```

# Hashtable for fast lookup

Dictionary-like data structure

Japan : Tokyo

USA : Washington

India : New Delhi

All operations  $O(1)$  on average\*:

- find
- insert
- erase

\*no worst case guarantees but fast in practice!

Blockchain | Likes | Caching | Passwords



Source: <https://medium.com/@saurabh.bhoy910/stl-unordered-map-hashtable-c7054b28d07f>

# Amazon User Tracking Scenario

Amazon engineers track ~200M unique users by IP addresses during peak shopping season. Each website access requires checking if the IP is in the list; if not, it's added. This helps analyze user interest and personalize recommendations.



Which data structure should Amazon engineers use to track ~200M unique users by IP addresses?

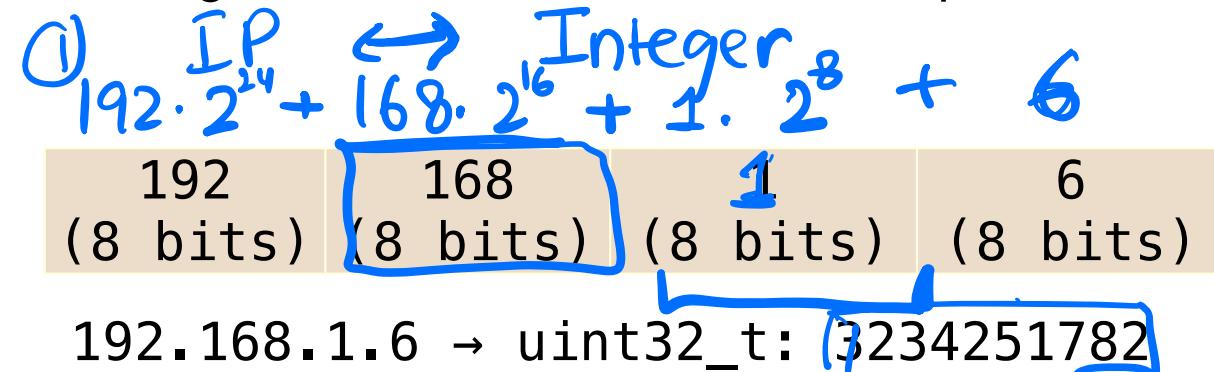
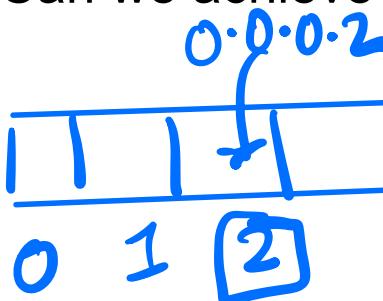
# Goal - Fast search (lookup) and insert

$0.0.0.2 \rightarrow 2$

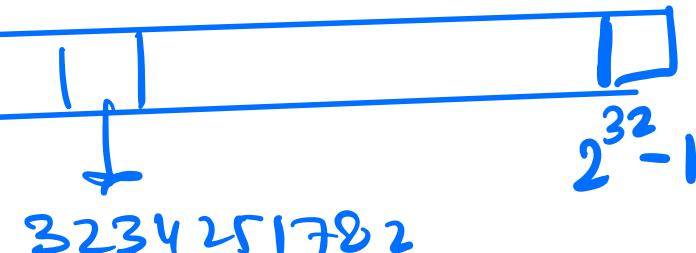
Which data structure should Amazon engineers use to track ~200M unique users by IP addresses?

- Keys: IP addresses (32 bits)
  - example : 192.168.1.6
- $2^{32} \sim 4.3B$  possible IPs
- 200M (4.7%) unique users

Can we achieve  $O(1)$  search?



$$(192 \ll 24 + 168 \ll 16 + 1 \ll 8 + 6)$$



# Naïve approach: $2^{32}$ -sized vector

*If you had to track unique IPs out of 4.3 billion ( $2^{32}$ ) possibilities, how much memory do you think you'd need if you used a direct index approach?*

$2^{10}$  bytes = 1KB,  $2^{20}$  bytes = 1 MB,  $2^{30}$  bytes = 1GB

Assuming 1 bit per IP, we need  $2^{32}$  bits =  $2^{29}$  bytes = 512 MB

Assuming 4 bytes per IP =  $512 * 32$  MB = 16 GB

# Setup for hash tables

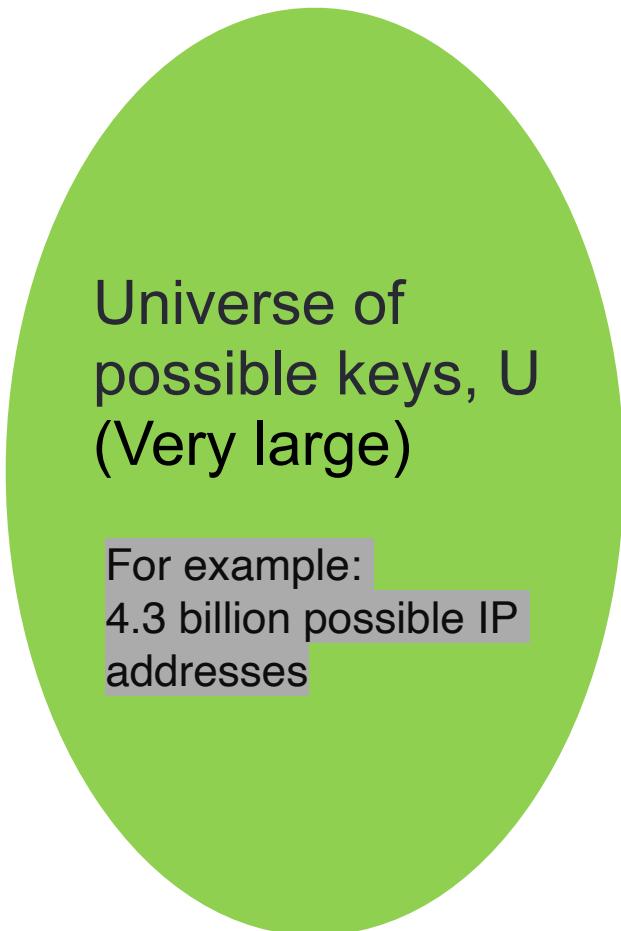
Universe of possible keys,  $U$   
(Very large)

For example:  
4.3 billion possible IP addresses

- Keep track of evolving set  $S$  whose size is much less than the universe of all possible keys
- For example, 200M unique users (~ 5% of all possible IPs)

0	
1	
2	Key
3	
4	
5	
6	
7	
8	
9	

# Insert key 192.168.1.6



Hash function:  $h(x)$   
Table size:  $m$   
Num keys:  $n$

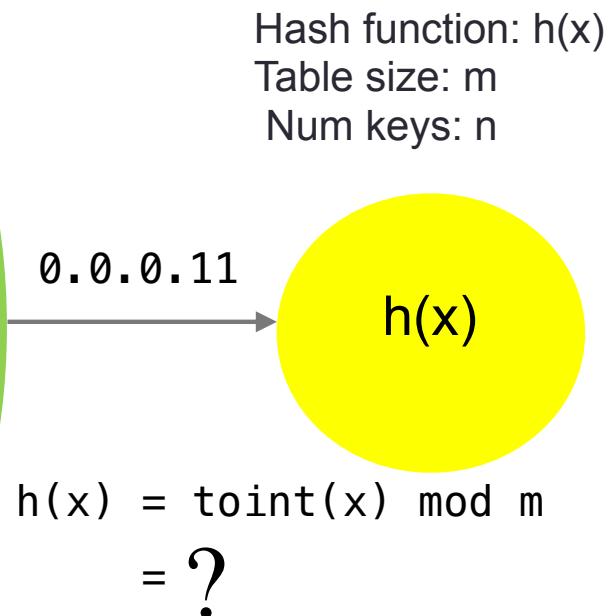
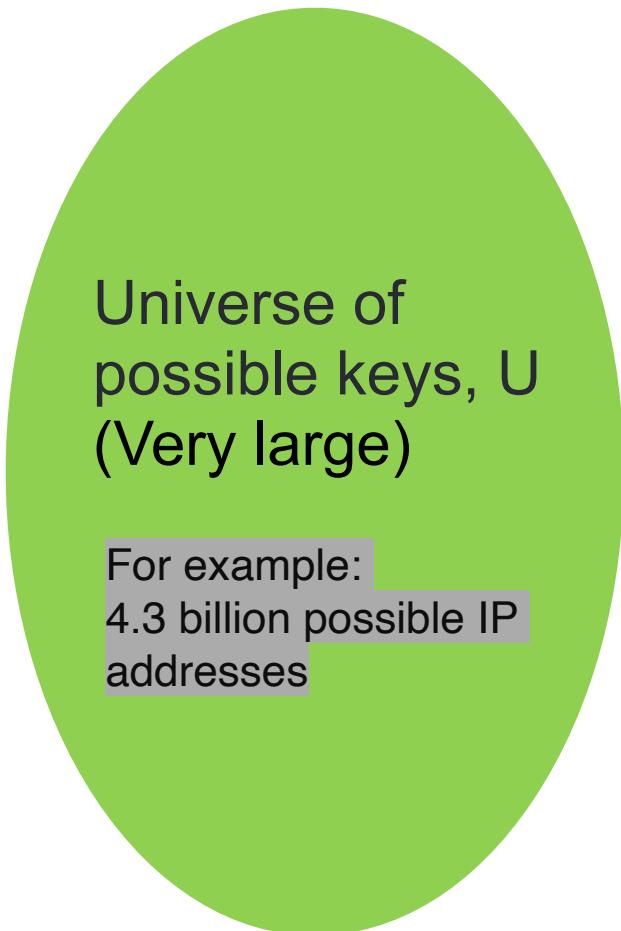
192.168.1.6

$$\begin{aligned} h(x) &= \text{toint}(x) \bmod m \\ &= \text{toint}(192.168.1.6) \bmod 10 \\ &= 3234251782 \bmod 10 \\ &= 2 \end{aligned}$$

- Keep track of evolving set  $S$  whose size is much less than the universe of all possible keys
- For example, 200M unique users (~ 5% of all possible IPs)

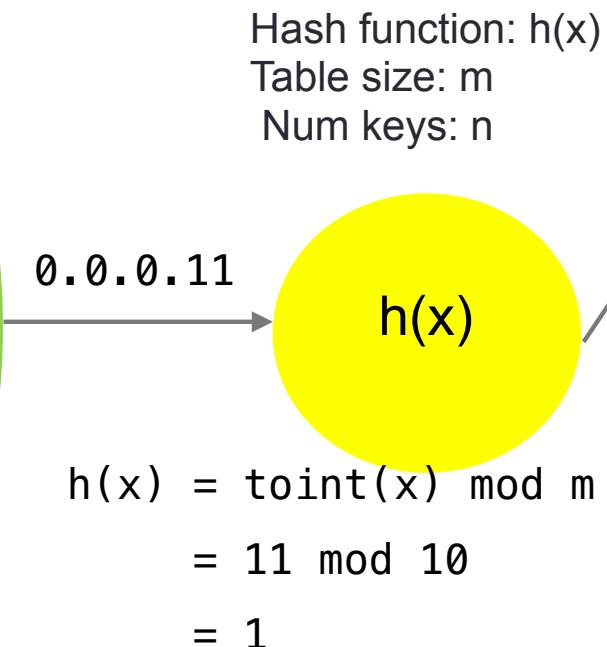
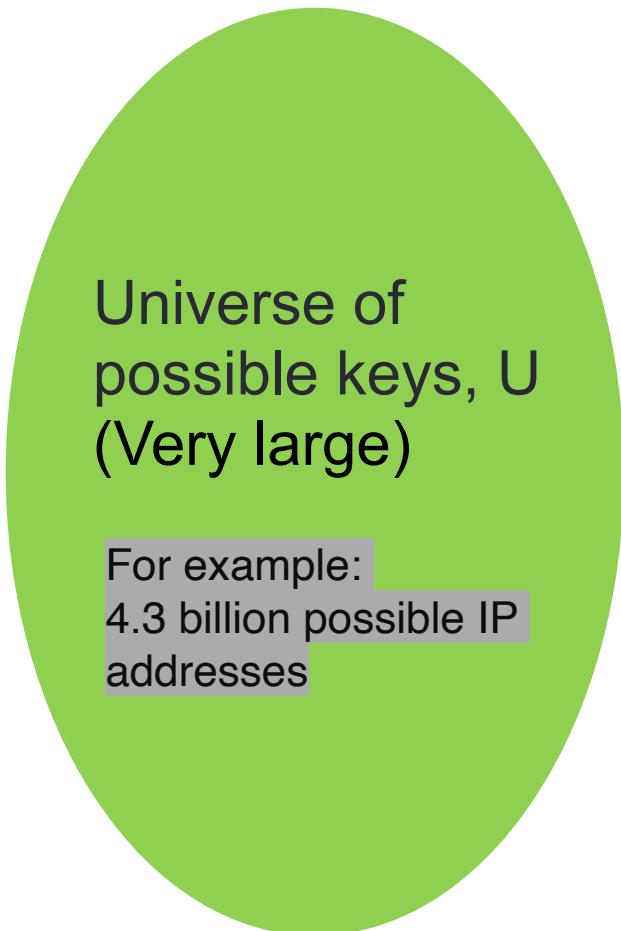
0	
1	
2	192.168.1.6
3	
4	
5	
6	
7	
8	
9	

# Insert key 0.0.0.11



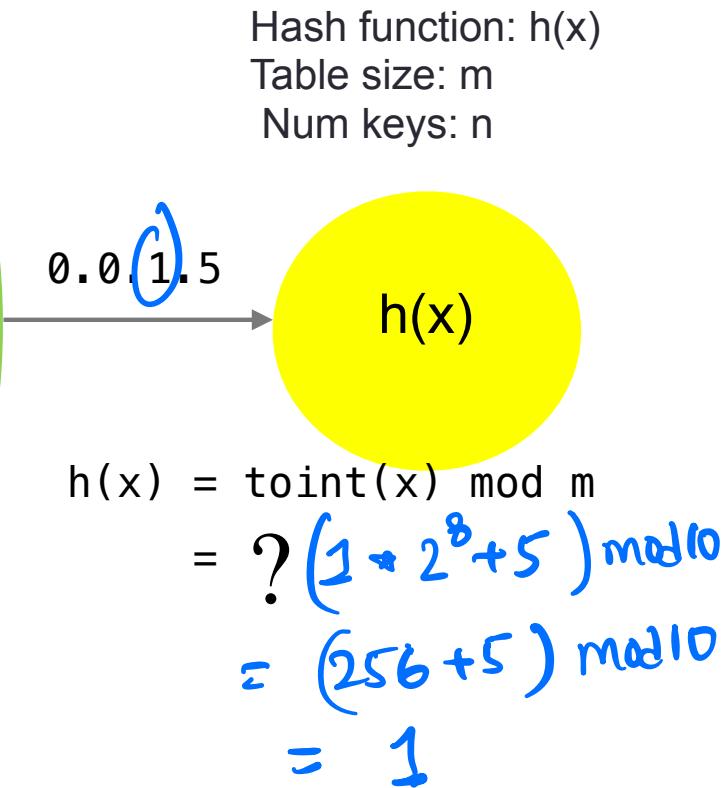
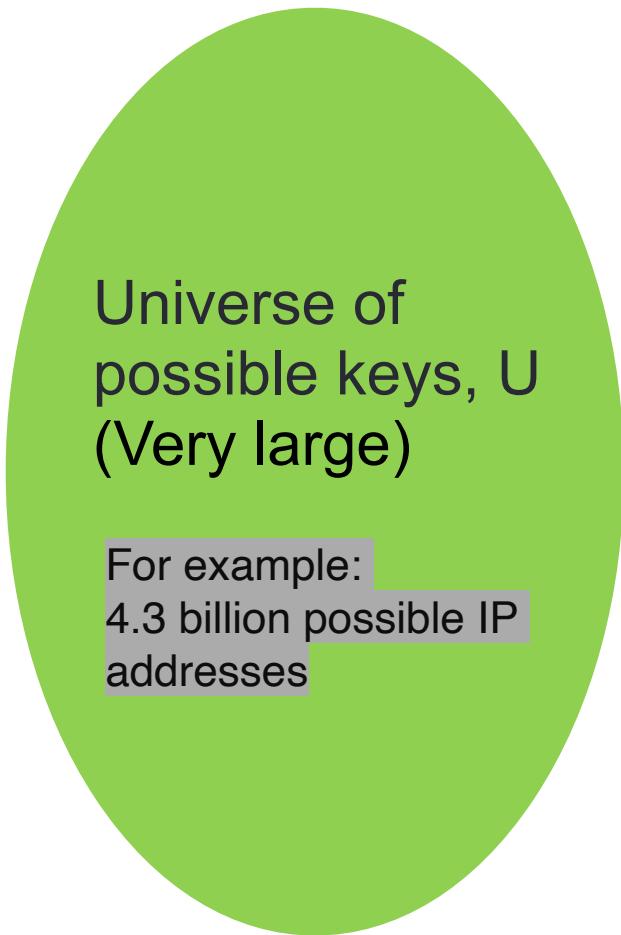
0	
1	
2	192.168.1.6
3	
4	
5	
6	
7	
8	
9	

# Insert key 0.0.0.11



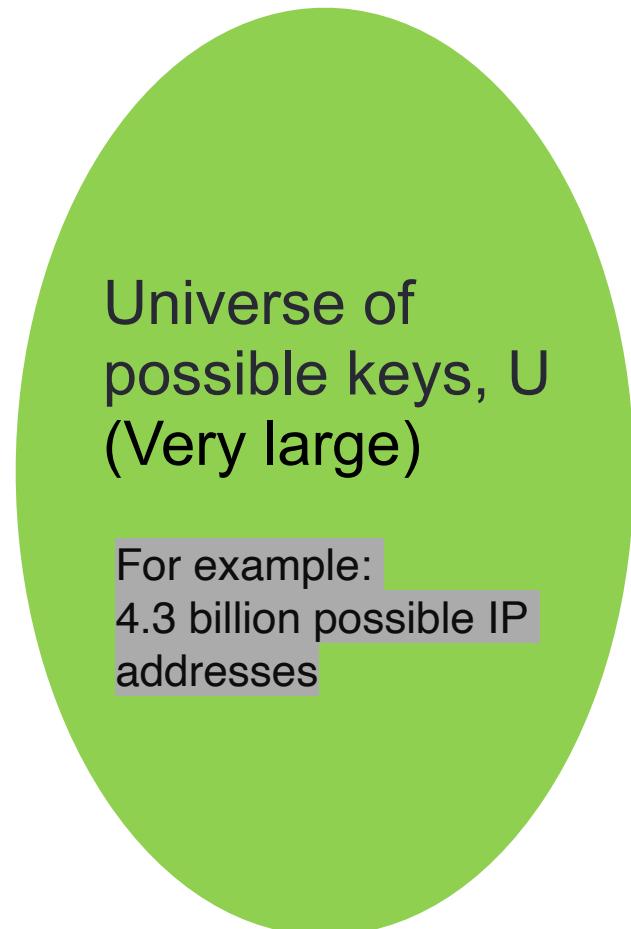
0	
1	0.0.0.11
2	192.168.1.6
3	
4	
5	
6	
7	
8	
9	

# Insert 0.0.1.5



0	
1	0.0.0.11
2	192.168.1.6
3	
4	
5	
6	
7	
8	
9	

# Insert key 0.0.1.5 results in a collision!



0.0.1.5

$$\begin{aligned} h(x) &= \text{toint}(x) \bmod m \\ &= (256 + 5) \bmod 10 \\ &= 261 \bmod 10 \\ &= 1 \end{aligned}$$

*What should happen when two IPs hash to the same index?*

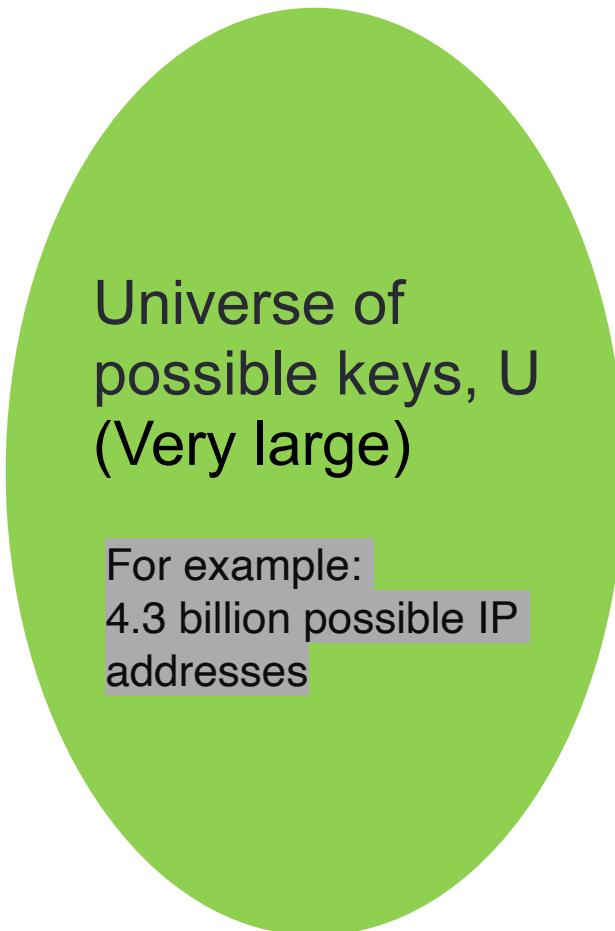
	A	B	C	D
Crash			Chain	Reject

Hash function:  $h(x)$   
Table size:  $m$   
Num keys:  $n$

Collision: Two keys map to the same spot!

0	
1	0.0.0.11
2	192.168.1.6
3	
4	
5	
6	
7	
8	
9	

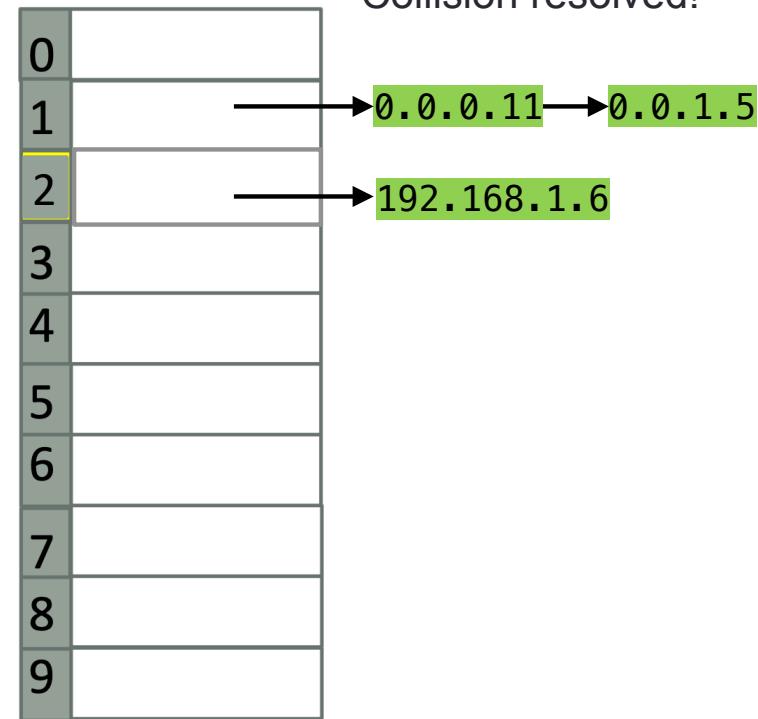
# Resolving collisions: separate chaining



0.0.1.5

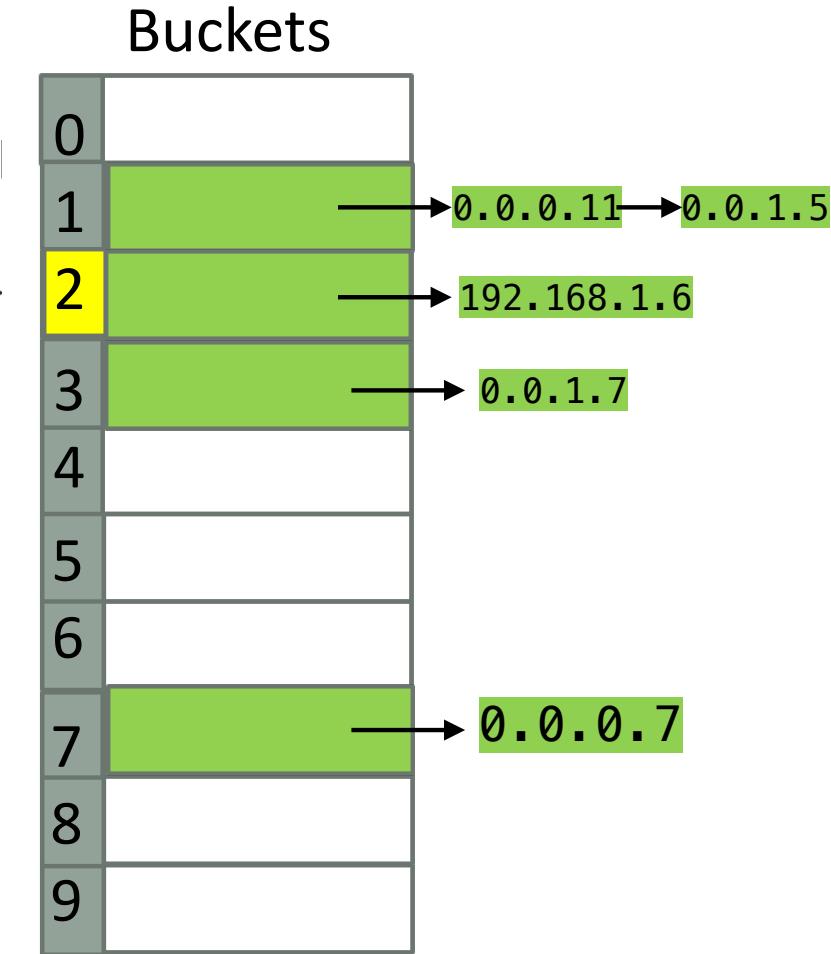
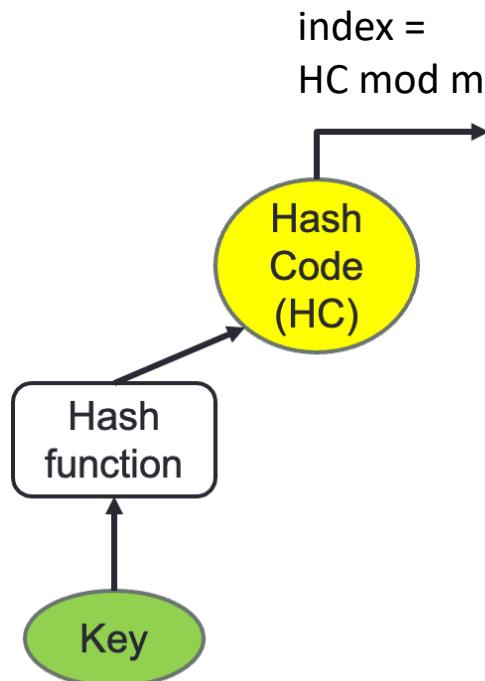
Hash function:  $h(x)$   
Table size:  $m$   
Num keys:  $n$

Buckets



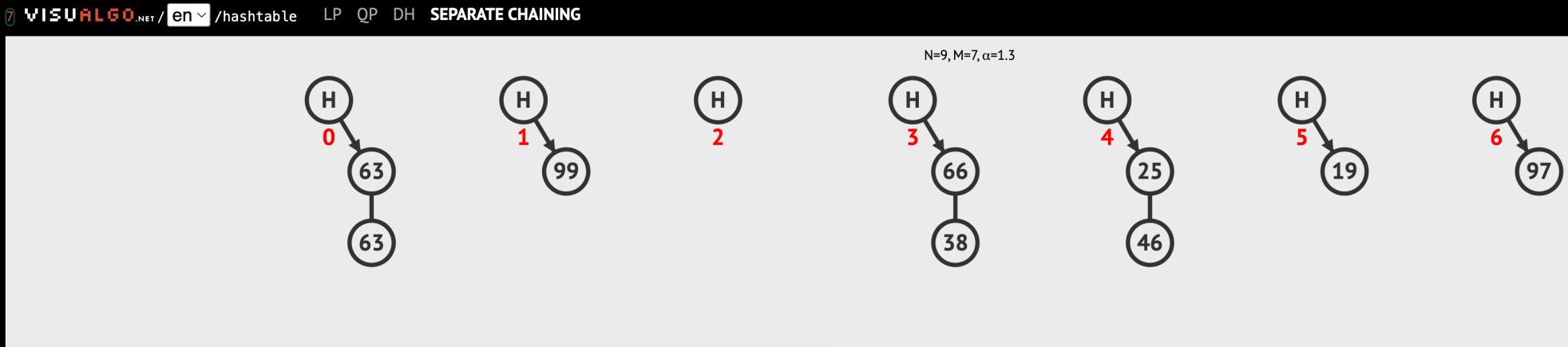
# (Refined) Logical model of a hash table

- Keys stored in buckets (vector)
- Keys used to compute index of position in vector
- Each bucket can store multiple keys as a linked list
- Hashtable with separate chaining: Vector of linked list



# Hashtable visualization

<https://visualgo.net/en/hashtable>



# Goal - Fast search (lookup) and insert

Which data structure should Amazon engineers use to track ~200M unique users by IP addresses?

- A) Set (Balanced BST)
- B) `unordered_set` (Hashtable)
- C) Priority Queue
- D) Queue
- E) Vector with  $2^{32}$  entries (one for each possible IP address)

192 (8 bits)	168 (8 bits)	1 (8 bits)	6 (8 bits)
-----------------	-----------------	---------------	---------------

192.168.1.6 → `uint32_t`: 3234251782

Total IPs:  $2^{32} \approx 4.3B$

# Design challenges

- Deciding on collision resolution strategy
- Deciding the size of hash table
- Deciding the hash function

Universe of possible keys,  $U$   
(Very large)

For example:  
4.3 billion possible IP addresses

Keep track of evolving set  $S$  whose size is much less than the universe of all possible keys

0	
1	
2	Key
3	
4	
5	
6	
7	
8	
9	

For example, 200M unique users  
(~ 5% of all possible IPs)

# Can we guarantee good performance?

We implemented a hashtable with separate chaining.

Table size:  $m$

Number of keys:  $n$

Load factor  $\alpha = n/m$

Hash function:  $h(x) = x \bmod m$

If randomly uniformly selected keys, then “on average” things seem fine.

Expected time to search is  $O(\alpha)$  (in practice, choose  $\alpha = 1$ )

**Can all keys still hash to the same bucket?  
(Worst-case scenario, linear search complexity!)**

# What's the chance of linear search complexity?

What is the probability that *all* keys hash to the *same* bucket?

Let's say we're inserting  $n = 1000$  keys into an empty table with  $m = 1000$  buckets.

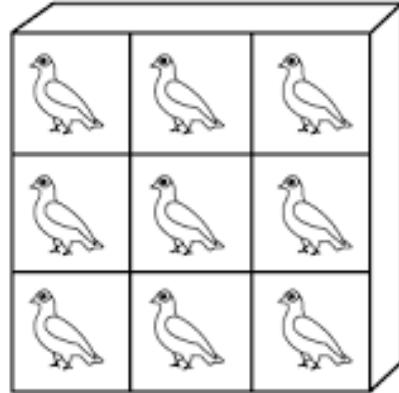
- A. 0
- B.  $1/1000$
- C.  $1/1000^{999}$
- D.  $1/1000^{1000}$
- E. It depends...

- We're using the hash function  $h(x) = x \bmod m$
- Keys are chosen independently and uniformly at random
- Hash table uses separate chaining

Is there a hash function that avoids linear search complexity?

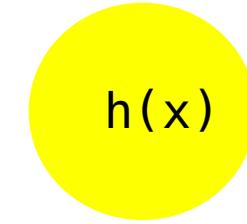
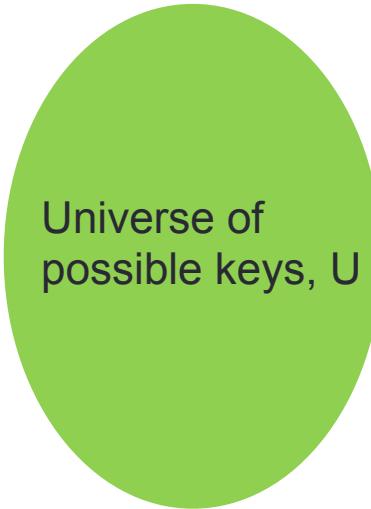
# Is there a hash function that avoids linear search complexity?

THE PIGEONHOLE PRINCIPLE



## Pigeonhole Principle:

If there are  $m$  pigeonholes and  $m + 1$  pigeons, at least one pigeonhole has more than 1 pigeon



## Generalized Pigeonhole Principle:

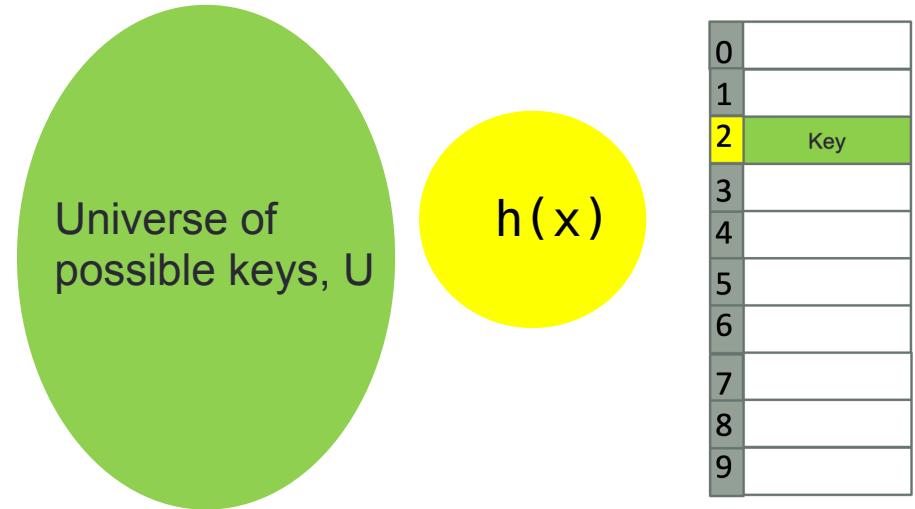
If there are  $m$  pigeonholes and  $nm + 1$  pigeons, at least one pigeonhole has more than \_\_\_\_\_ pigeons.

# Is there a hash function that avoids linear search complexity?

**Theorem:** Suppose a hash table of size  $m$  is used to store a set  $S$  of  $n$  keys drawn from the universe  $U$ , where  $|U| > nm$ . Then, *no matter which hash function  $h : U \rightarrow \{0,1,2,\dots,m-1\}$  is chosen, there is a set  $S \subset U$  of  $n$  keys that all map to the same location.*

## Proof:

1. Pick any hash function of your choosing.
2. Map all keys of  $U$  using  $h$  to the table of size  $m$ .
3. By the pigeonhole principle, at least one table slot gets at least  $n$  keys because  $|U| > nm$
4. Choose those  $n$  keys as input set  $S$
5. Now  $h$  will map  $S$  to a single location.



**Generalized Pigeonhole Principle:**  
If there are  $m$  pigeonholes and  $nm + 1$  pigeons, at least one pigeonhole has more than  $n$  pigeons.

What does this mean for real-world applications?

# Universal Hash Functions

**Main result so far:** No single hash function can avoid worst case linear time search complexity!

**Main idea behind universal hash functions:** Don't fix a single hash function. Choose one randomly from a “good” family of hash functions.

- Imagine an adversary picks **two keys**,  $x$  and  $y$ .
- You, the algorithm designer, get to **randomly choose** a hash function  $h$  from a big family  $H$ .
- The hash function is designed so that:  
The chance that  $h(x) = h(y)$  is **at most  $1/m$** .

Example:  $h(x) = ax + b \pmod{p}$  where  $p$  is a prime number

# References

**Professor Subhash Suri's CS 130A handout on hash tables:**

<https://sites.cs.ucsb.edu/~suri/cs130a/Hashing.pdf>