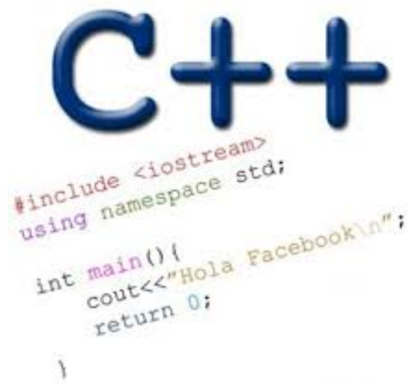
FAST LOOKUP WITH HASHTABLES

Problem Solving with Computers-II

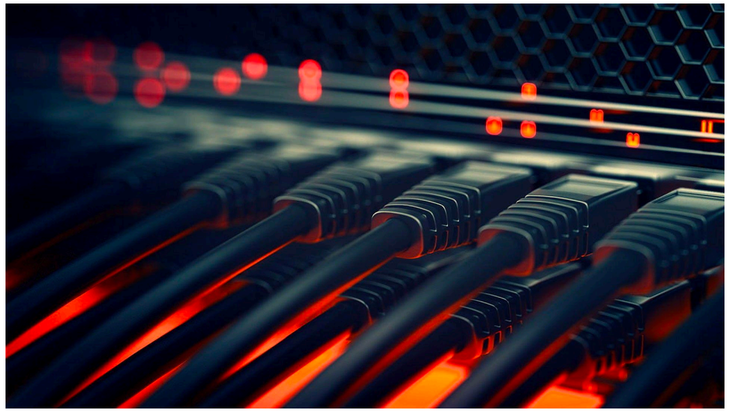
STD::UNDORDERED_SET
STD::UNORDERED_MAP



```
#include <iostream>
using namespace std;

int main(){
    cout<<"Hola Facebook\n";
    return 0;
}
```

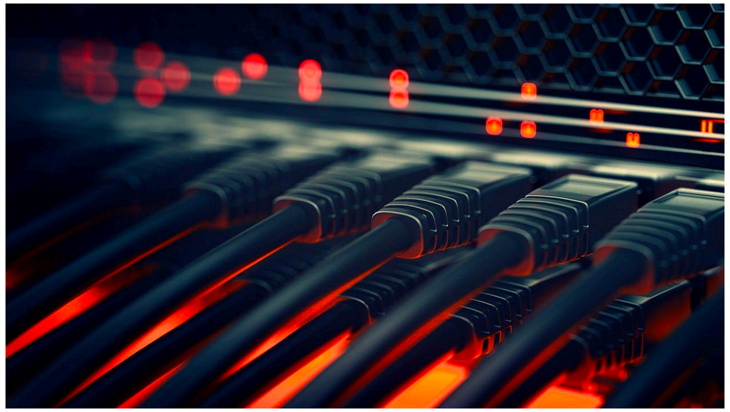
Hash table: Practical Applications



Network security

- Routers process 2-3 million packets per second
- Security strategies include:
  - Dropping packets from blacklisted sources
  - Need to lookup IP address in blacklist

Hash table: Practical Applications



Amazon makes approx. $1.3 billion per day.

In July 2020, Amazon sites had 213 million unique visitors in the US.

Network security

- Routers process 2-3 million packets per second
- Need efficient lookup of IP address in a blacklist

Website analytics

- How many unique visitors to a website?
- Need efficient way to de-duplicate entires in very large logs

Many applications rely on fast lookup in an evolving dataset

Last time: Balanced BST

```
set<string> groceries;
groceries.insert(“Banana”) ;
groceries.insert(“Apple”);
groceries.insert(“Milk”);
groceries.insert(“Bread”);
```

```
map<string, int> groceries;
groceries[“Banana”] = 2;
groceries[“Apple”] = 1;
groceries[“Milk”] = 3;
groceries[“Bread”] = 5;
```

std::set
stores unique keys

std::map
stores key, value pairs

Today: Hash table

```
unordered_set<string> groceries;
groceries.insert("Banana") ;
groceries.insert("Apple");
groceries.insert("Milk");
groceries.insert("Bread");
```

std::unordered_set

stores unique keys

```
unordered_map<string, int> groceries;
groceries["Banana"] = 2;
groceries["Apple"] = 1;
groceries["Milk"] = 3;
groceries["Bread"] = 5;
```

std::unordered_map

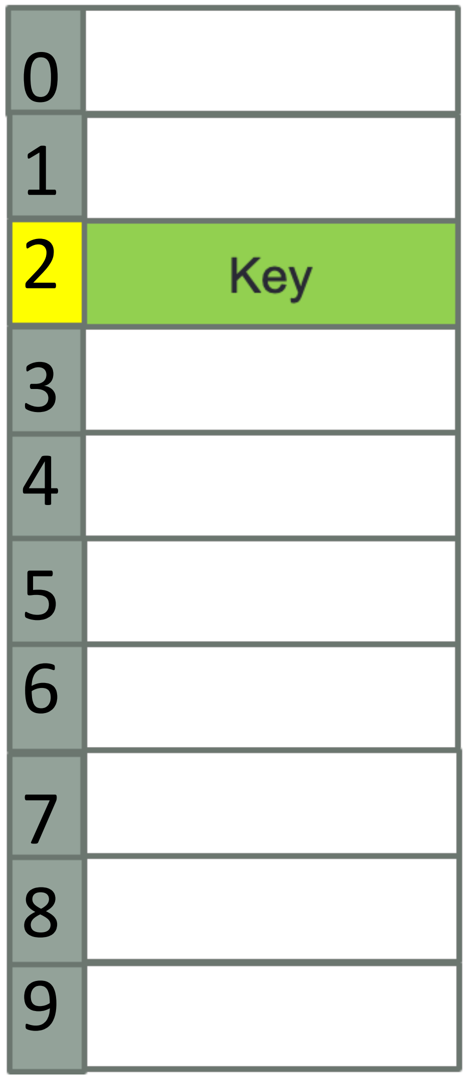
stores key, value pairs

Operations: find, insert, erase: all O(1)

- **not a worst case guarantee**
- **only if the hash table is implemented properly**

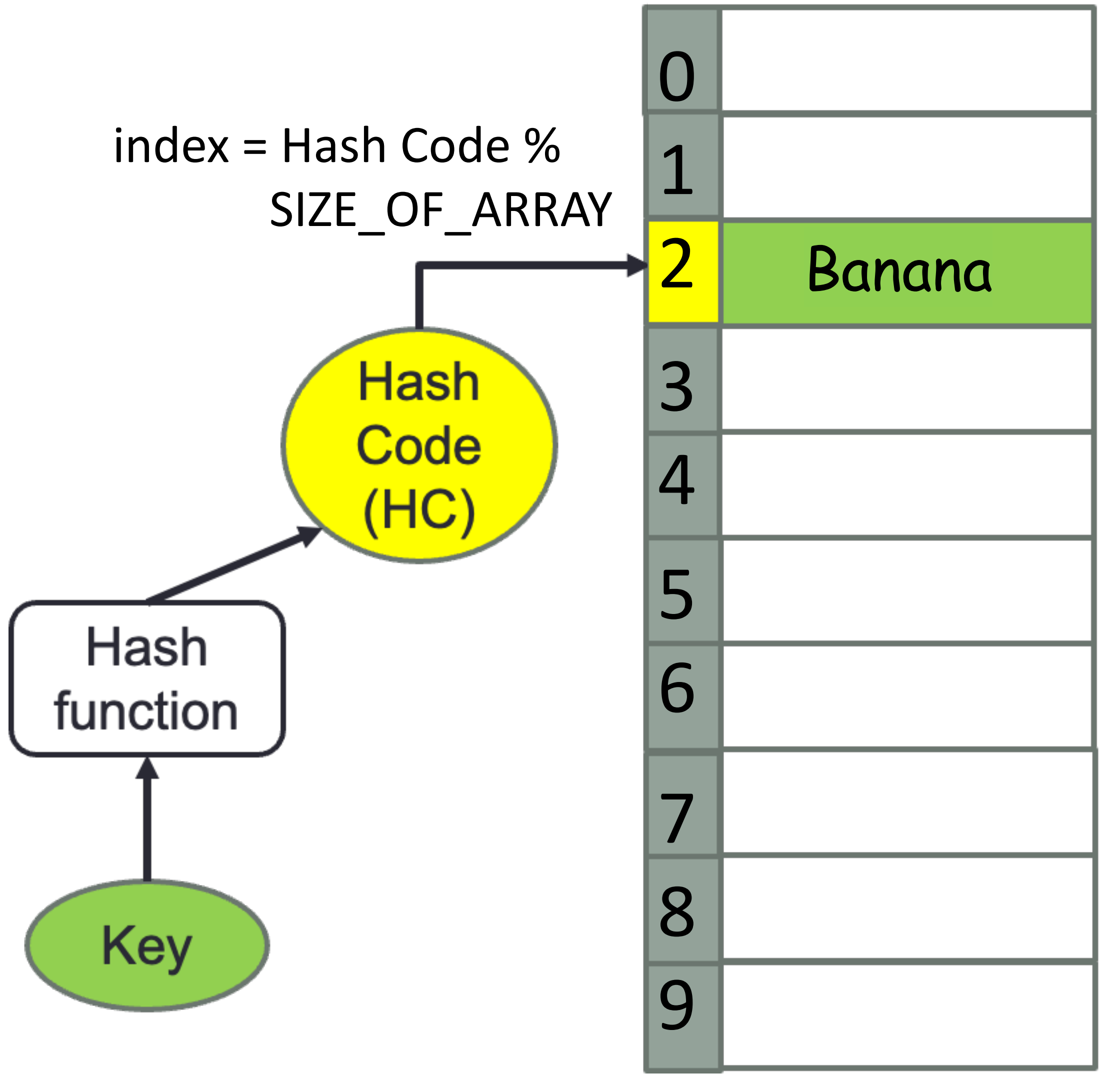
Hash table: an array with positions indexed by keys

```
unordered_set<int> groceries;
groceries.insert(2) ;
groceries.insert(1);
groceries.insert(7);
groceries.insert(3);
groceries.find(2);
```



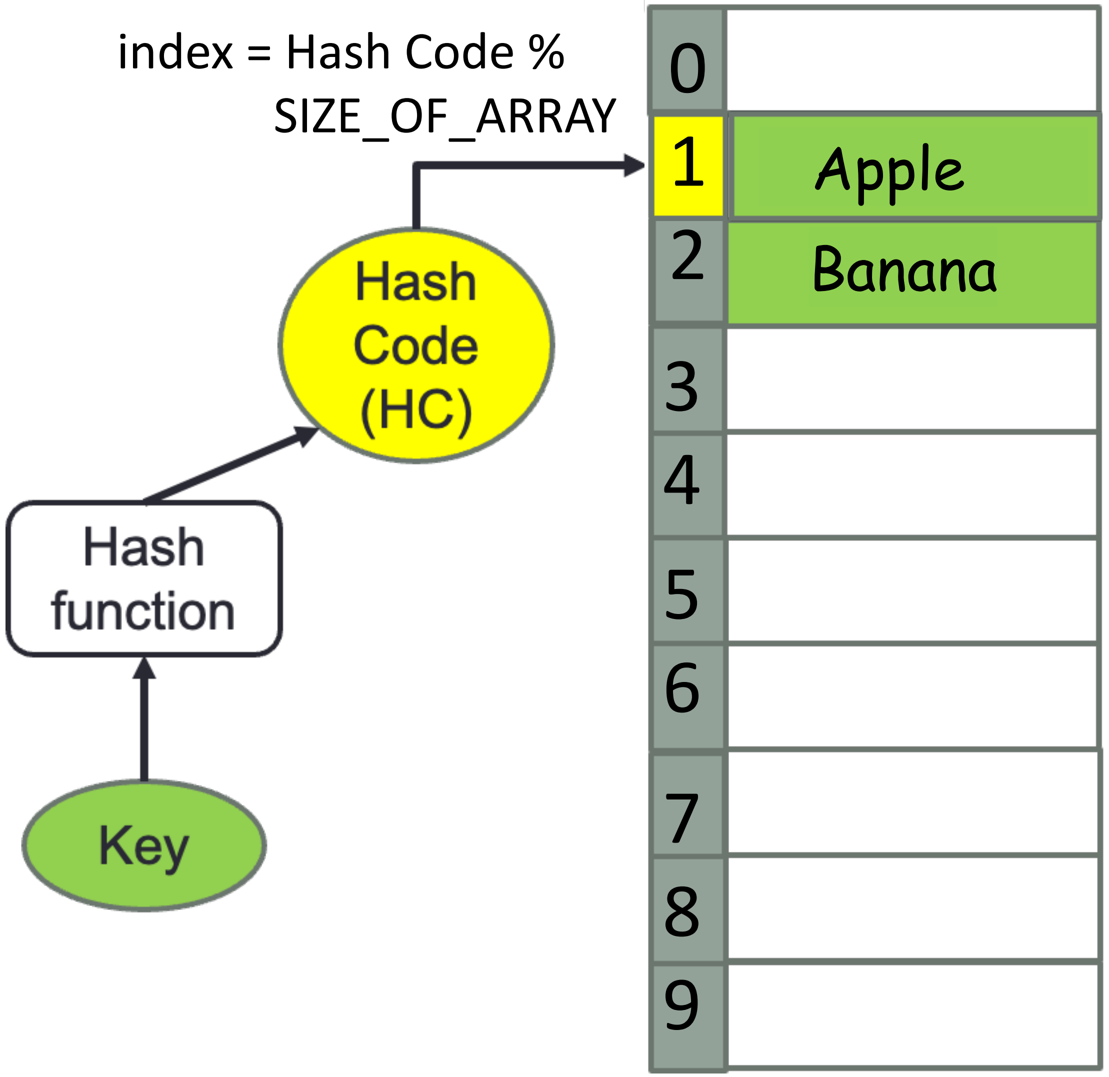
Hash table: an array with positions indexed by keys

```
unordered_set<string> groceries;
groceries.insert("Banana");
```



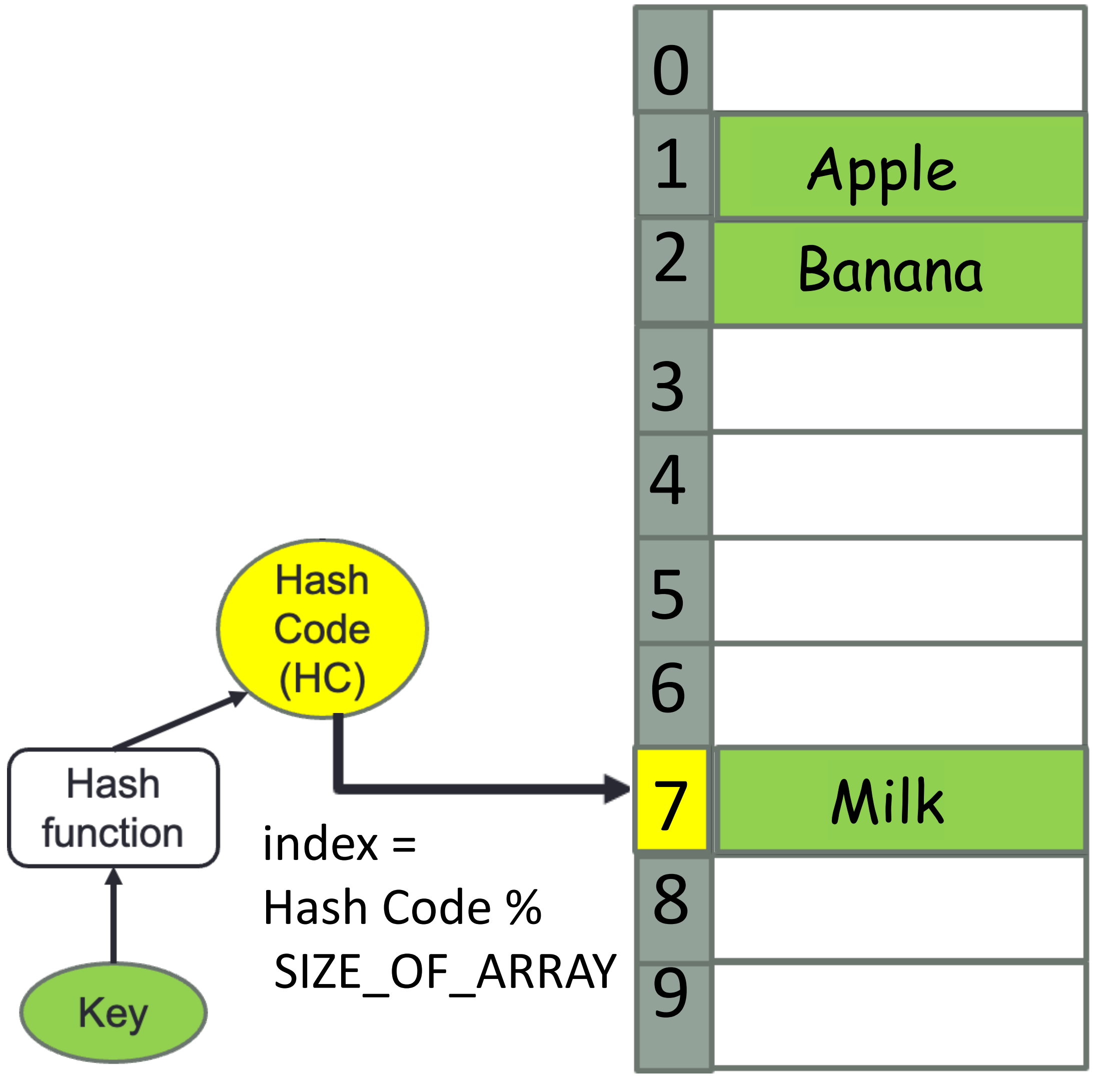
Hash table is just an array with positions indexed by keys

```
unordered_set<string> groceries;
groceries.insert("Banana");
groceries.insert("Apple");
```



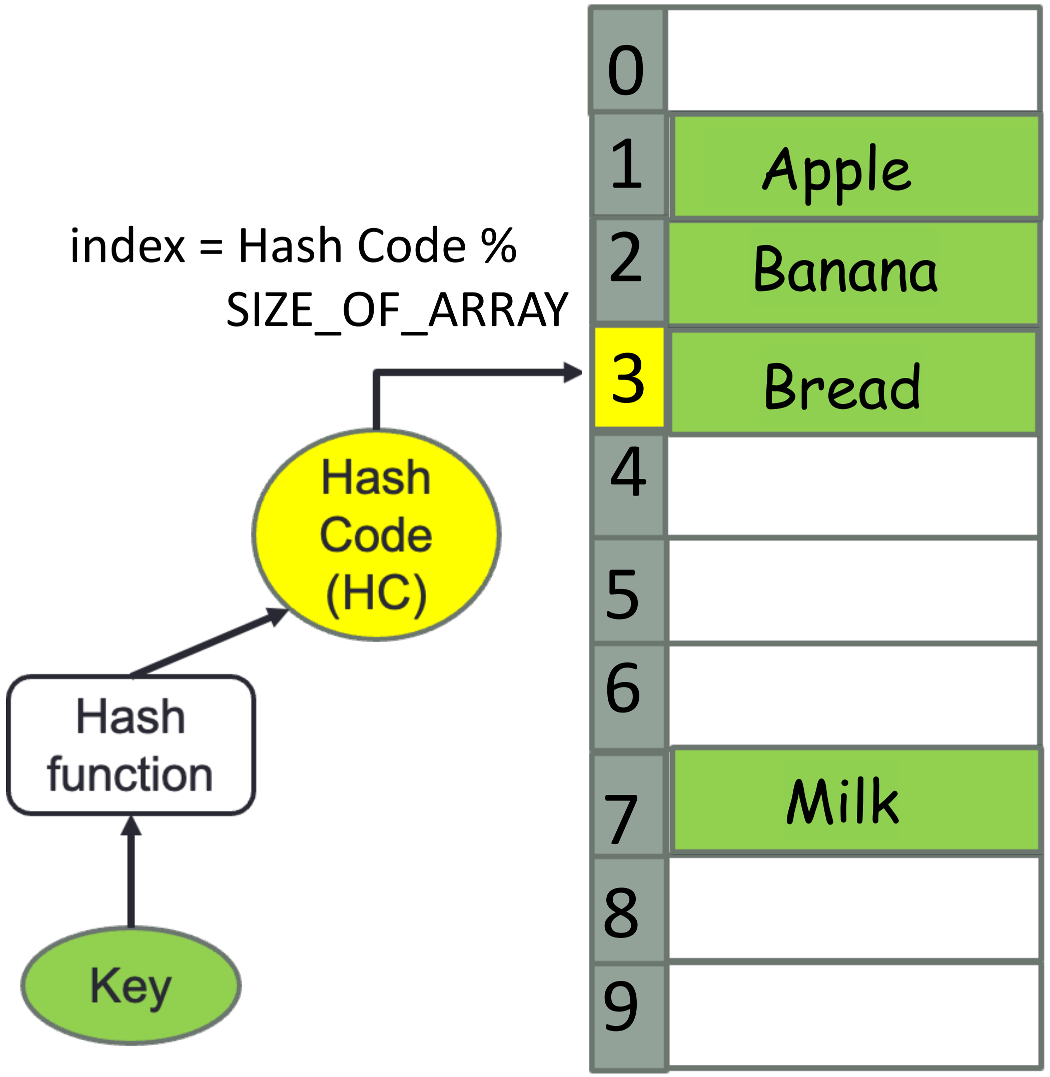
Hash table is just an array with positions indexed by keys

```
unordered_set<string> groceries;
groceries.insert("Banana");
groceries.insert("Apple");
groceries.insert("Milk");
```



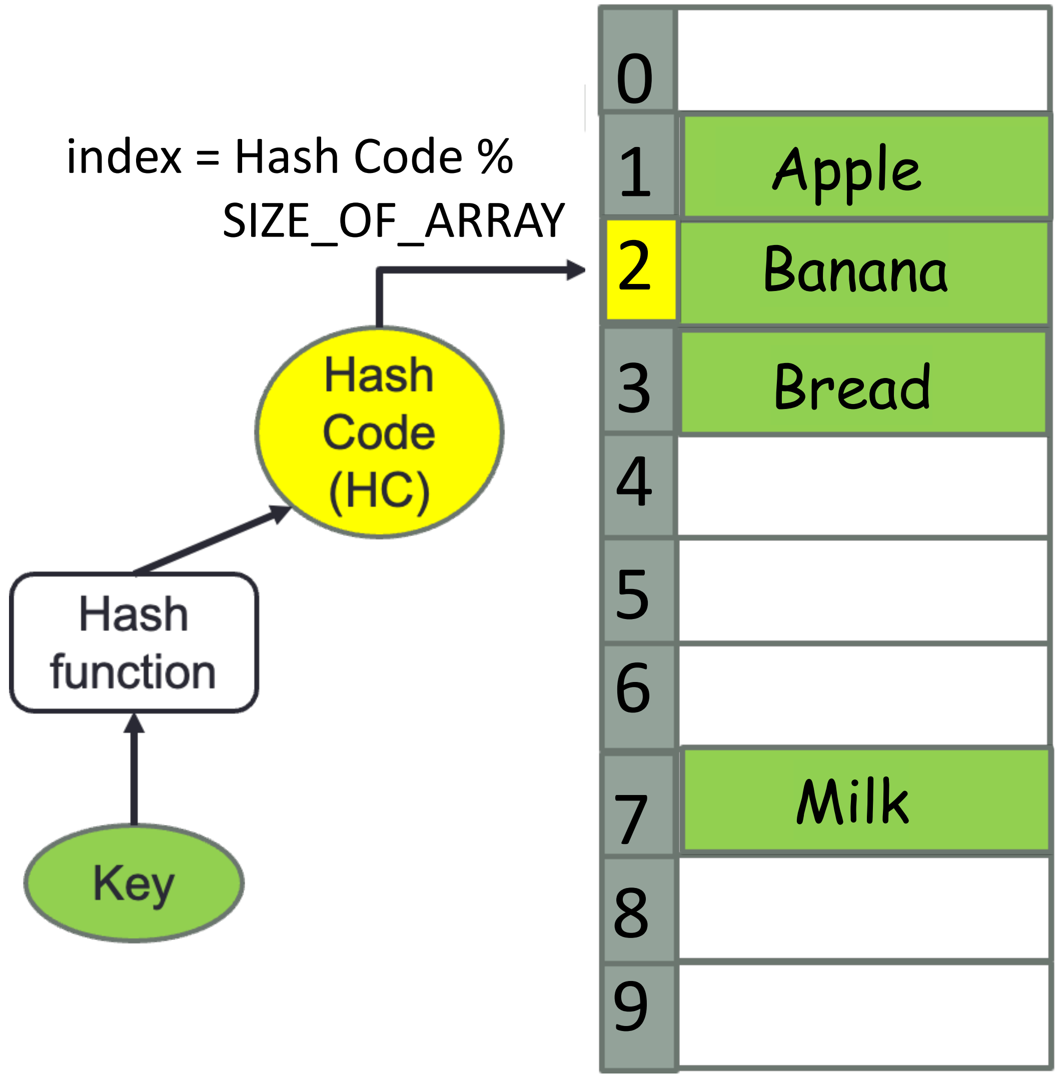
Hash table is an array with positions indexed by keys

```
unordered_set<string> groceries;
groceries.insert("Banana");
groceries.insert("Apple");
groceries.insert("Milk");
groceries.insert("Bread");
```



Hash table is an array with positions indexed by keys

```
unordered_set<string> groceries;
groceries.insert("Banana");
groceries.insert("Apple");
groceries.insert("Milk");
groceries.insert("Bread");
groceries.find("Banana");
```

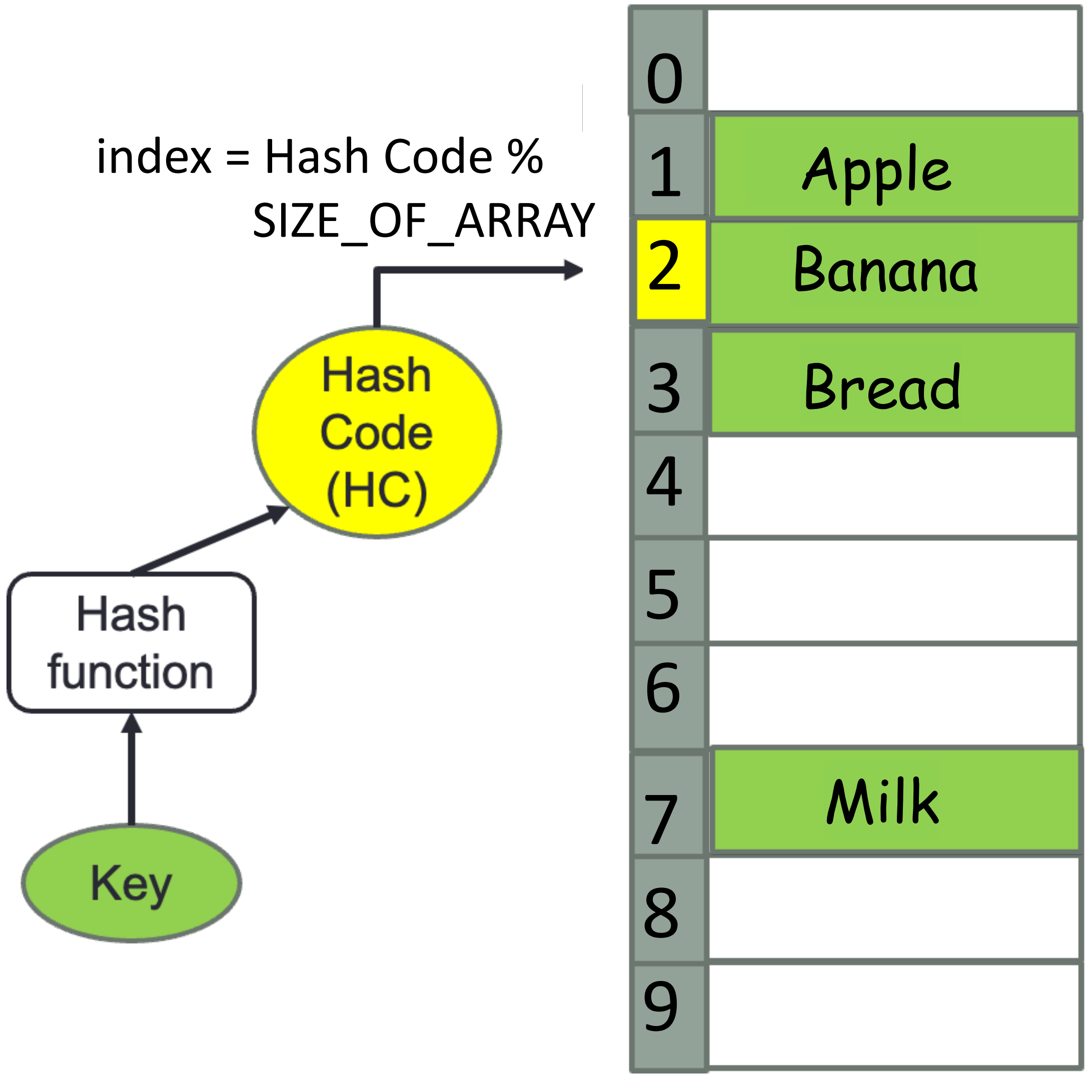


Hash table is an array with positions indexed by keys

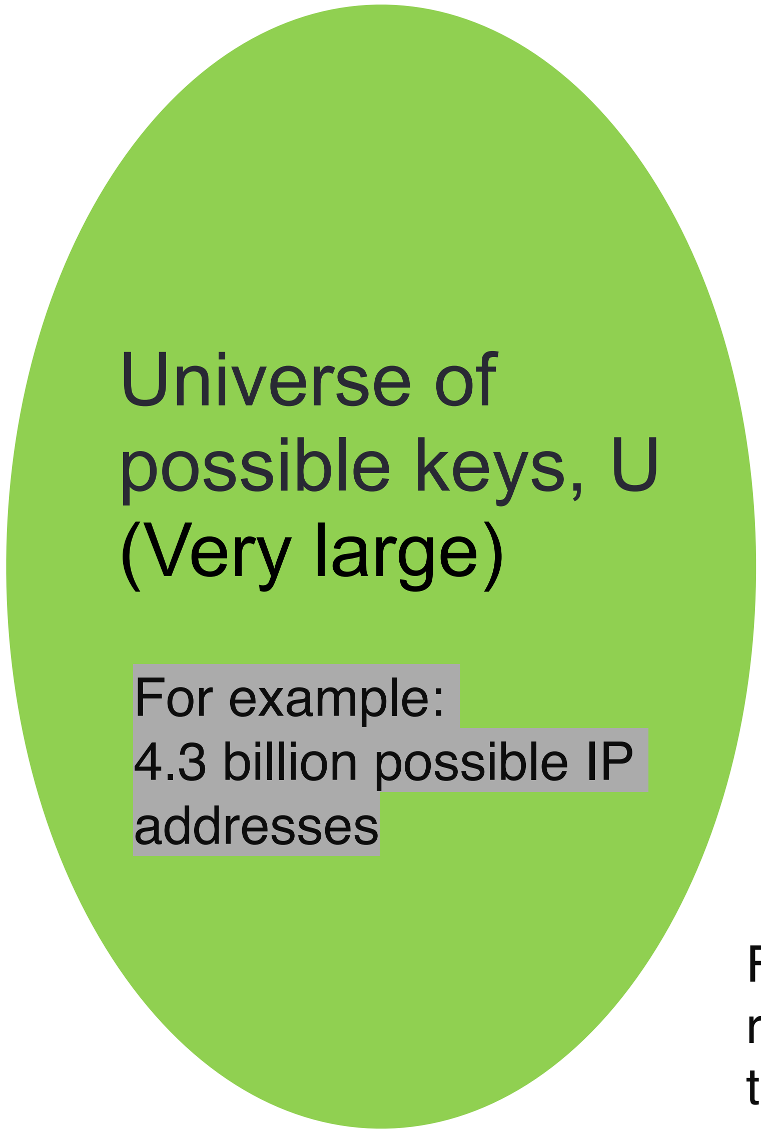
```
unordered_set<string> groceries;
groceries.insert(“Banana”);
groceries.insert(“Apple”);
groceries.insert(“Milk”);
groceries.insert(“Bread”);
groceries.insert(“Grape”);
```

Suppose our hash function outputs the hash code 12 for “Banana” and 32 for “Grape”, what happens when we try to insert “Grape”?

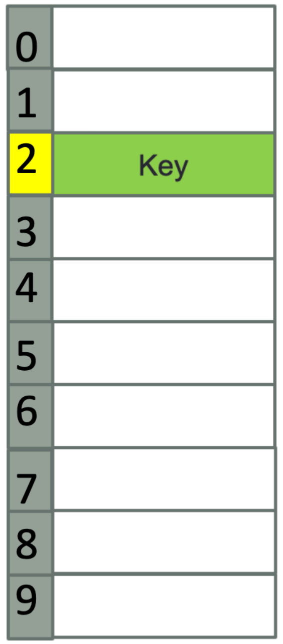
A. “Grape” cannot be inserted
B. “Banana” should be replaced by “Grape”
C. Both “Banana” and “Grape” map to the same index, resulting in a collision



Setup for hashing


Universe of possible keys, U
(Very large)
For example:
4.3 billion possible IP addresses

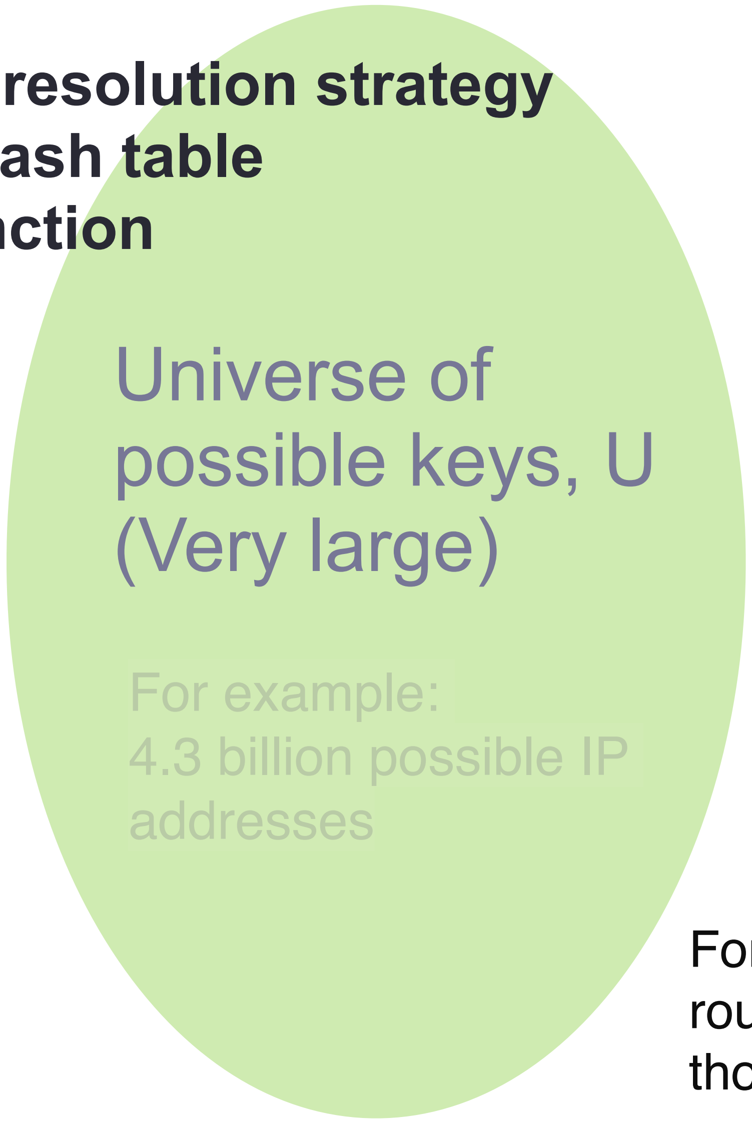
Keep track of evolving set S whose size is much less than the universe of all possible keys



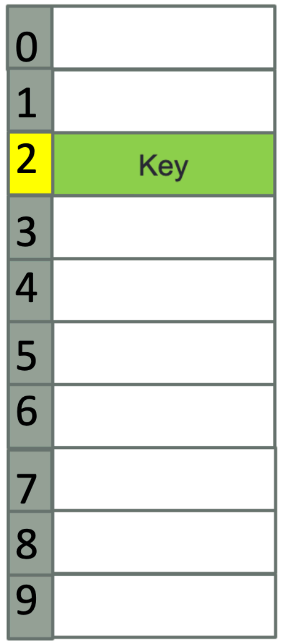
For example, size of a blacklist on an internet router is typically a few hundred to tens of thousands of entries

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

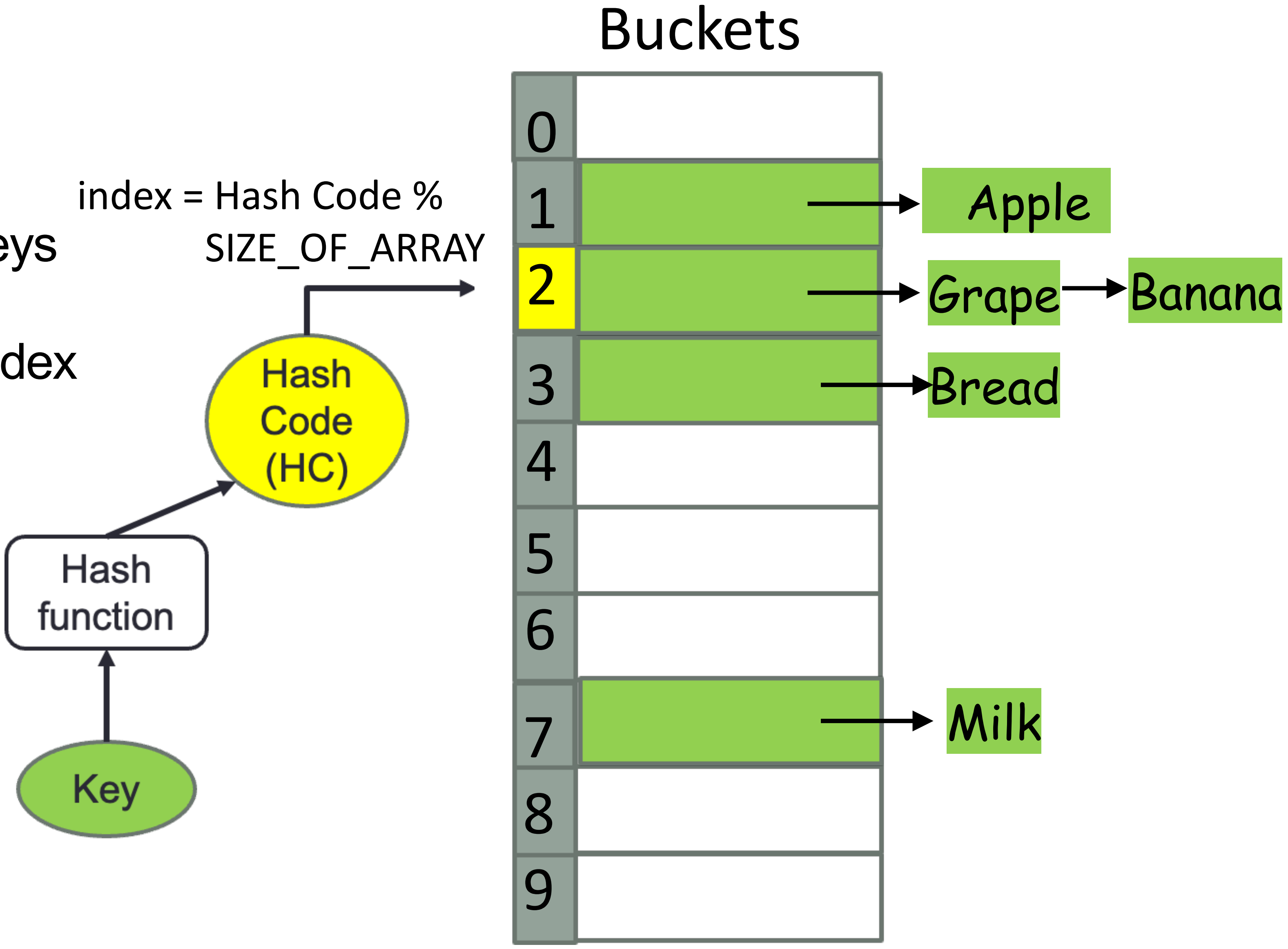
Keep track of evolving set S whose size is much less than the universe of all possible keys



For example, size of a blacklist on an internet router is typically a few hundred to tens of thousands of entries

(Refined) Logical model of a hash table

- Keys stored in buckets (array)
- Array positions are indexed by keys
- Multiple keys map to the same index



Two sum problem

Given an unsorted array (A) of N unique integers between 0 and 1000,000, find the pair of elements that sum to a given number T (assume a unique solution exists)

Method 1: Exhaustive search

```
procedure twosum(A: array of integers of size N, T: target sum)
  pair<int, int> result
  for (i = 0; i < N; i++){
     for (j = i+1; j < N; j++){
         if (A[i] + A[j]) == T
            result = (A[i], A[j]);
       }
  }
  return result {pairs of values that add to T}
```

What is the worst case run time of this method?

A. O(N)
B. (N logN)
C. O(N^2)

Two sum problem

Given an unsorted array (A) of N unique integers between 0 and 1000,000, find all pairs of elements that sum to a given number T (assume unique solution exists)

Method 2: Sort, then search

```
procedure twosum(A: array of integers of size N, T: target sum)
  pair<int, int> result
  sort(A)
  for (i = 0; i < N; i++){
     if(binary_search(T – A[i]))
        result = (A[i], T – A[i])
  }
  return result {pairs values that add to T}
```

What is the worst case run time of this method?

A. O(N)
B. (N logN)
C. O(N^2)

Two sum problem

Given an unsorted array (A) of N unique integers between 0 and 1000,000, find all pairs of elements that sum to a given number T

Method 3: Use hash tables

```
procedure twosum(A: array of integers of size N, T: target sum)
  pair<int, int> result









return result {pairs of values add to T}
```

Discuss a method that uses hash tables. What is the worst case run time of this method?

A. O(N)
B. (N logN)
C. O(N^2)

Suppose you have a hash table that can hold 100 elements. It currently stores 9 elements (in 9 different locations in the hash table). What is the probability that your next insert will cause a collision?

Assume a hash function that maps keys to slots with equal likelihood

A. 0
B. 9/100
C. 50/100
D. 74/100
E. 1

Suppose you have a hash table that can hold 100 elements. It currently stores 30 elements (in one of 30 possible different locations in the hash table). What is the probability that your next two inserts will cause at least one collision?

Assume a hash function that maps keys to slots with equal likelihood.

A. 0
B. 9/100
C. 50/100
D. 74/100
E. 1

Consider a party with n people, each person’s birthday is on any day of the year with equal likelihood. How large does *n* need to be before there is a 50% chance that at least two people have the same birthday.

A. 27
B. 50
C. 60
D. 200
E. 366

Suppose there are 365 slots in the hash table: M=365
What is the probability that there will be a collision when inserting N keys?
For N = 10, probN,M(collision) = 12%
For N = 20, probN,M(collision) = 41%
For N = 30, probN,M(collision) = 71%
For N = 40, probN,M(collision) = 89%
For N = 50, probN,M(collision) = 97%
For N = 60, probN,M(collision) = 99+%

So, among 60 randomly selected people, it is almost certain that at least one pair of them have the same birthday

On average one pair of people will share a birthday in a group of about
$\sqrt{2 * 365} = 27$ people
In general: collisions are likely to happen, unless the hash table is quite sparsely filled

Probability of collision

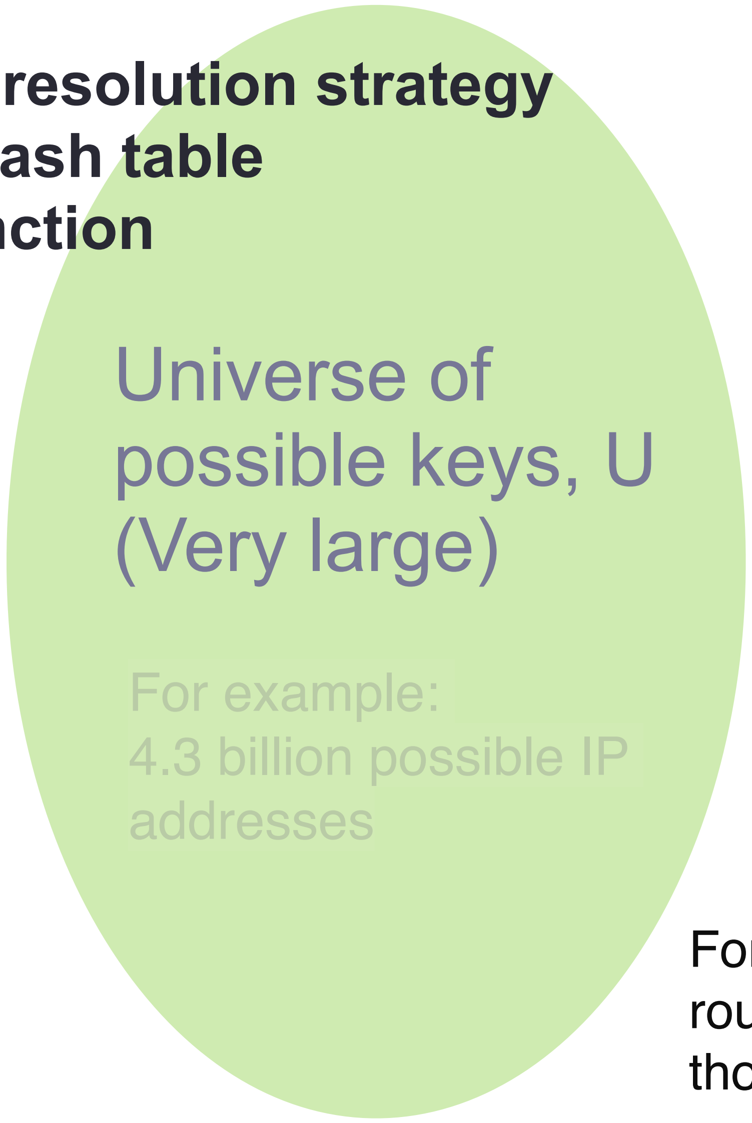
If a hash table has M slots and N keys, assuming your hashing function map keys to each slot with equal likelihood, the probability of at least one collision is

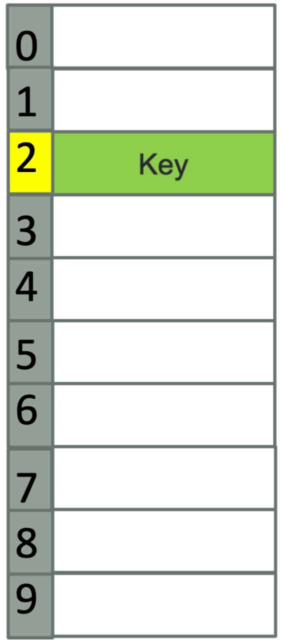
$$P_{N,M}(collision) = 1 - \prod_{i=1}^{N} \frac{(M - (i - 1))}{M}$$

Hash table design (will be covered in depth in 130A)

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

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


Universe of possible keys, U (Very large)
For example: 4.3 billion possible IP addresses



For example, size of a blacklist on an internet router is typically a few hundred to tens of thousands of entries