FAST LOOKUP WITH HASHTABLES

Problem Solving with Computers-II

Amazon IP Tracking Problem

Imagine you're a software engineer at Amazon. It's Prime Day. Your team is responsible for tracking unique visitors to the site.

Here's your problem: 200 million users are hitting the site today. Every single page request, you need to check if this IP address has visited before. If not, add it to your set of known visitors.

Your code runs on every. single. request. It needs to be fast!

What data structure do you use?

Open `traffic_activity.cpp`

U: Universe of Keys

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

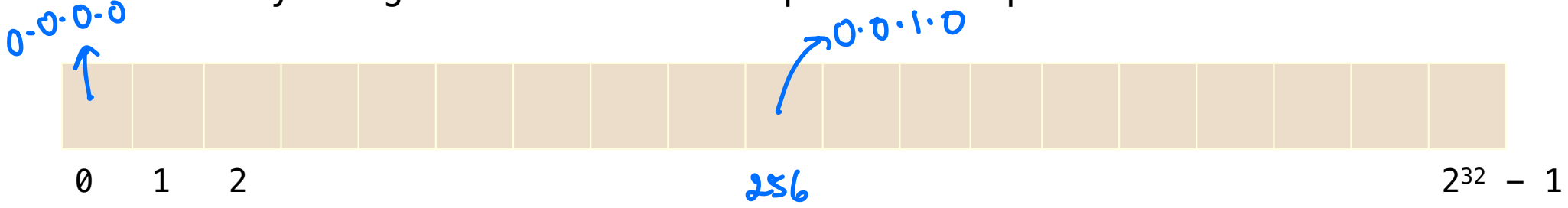
An IP is 32 bits → 4.3B (2^32) possible IPs

O(1) worst case….at a cost

0.0.0.1 → 1
0.0.1.0 → 256

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

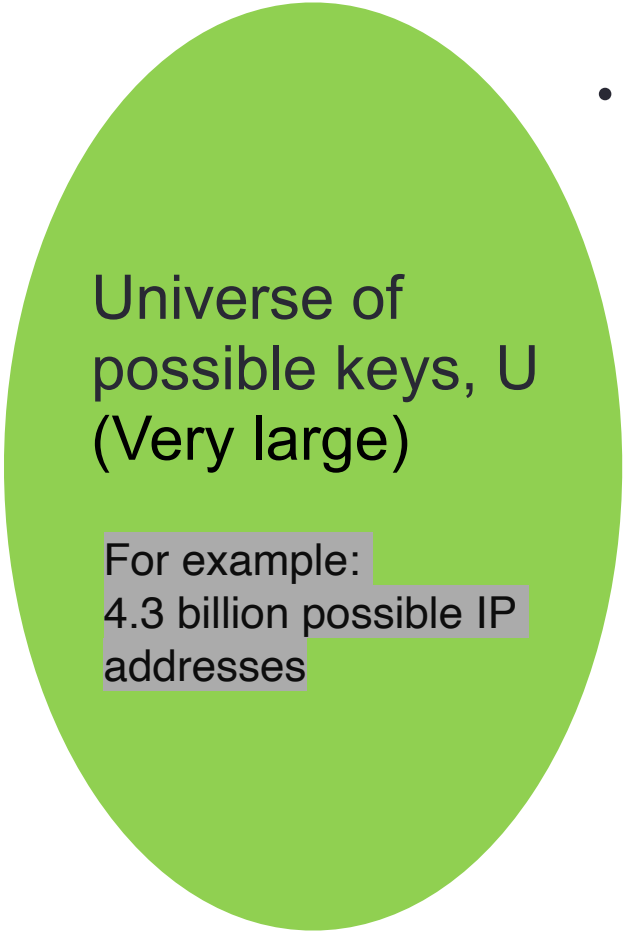
Use a very large vector with 2^{32} possible spots



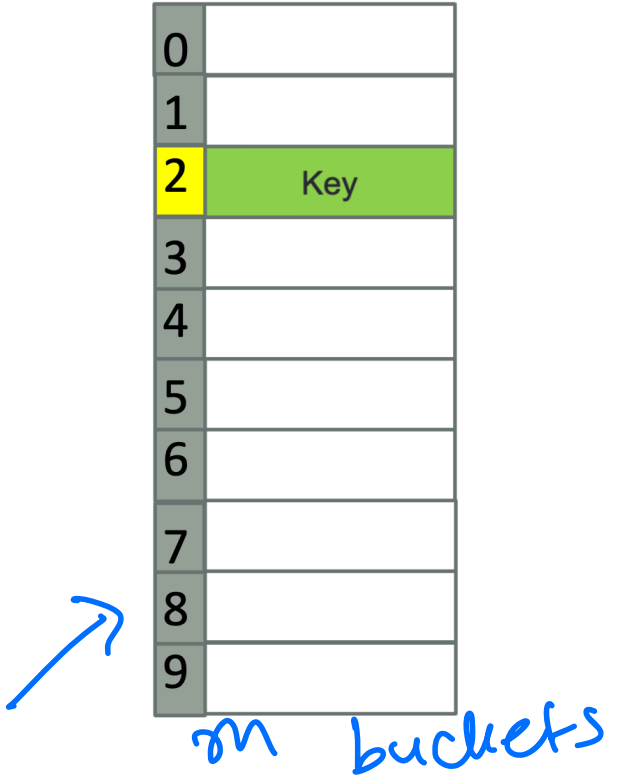
First approach: Use key value (IP) as index in a 2^{32} sized vector

But you're only tracking 200 million: about 5% of the space
So, this method wastes space.

Setup for hash tables



- 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)

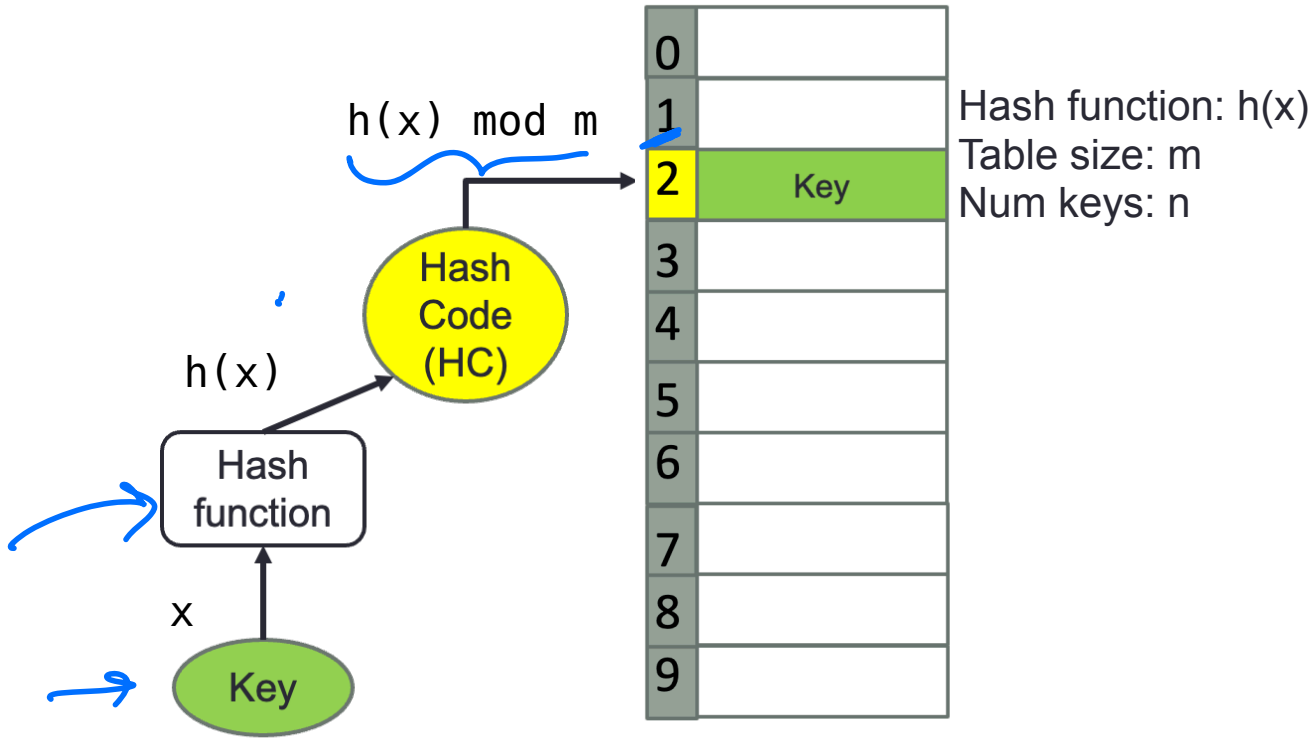


Setup for hash tables

Universe of possible keys, U (Very large)

For example:
4.3 billion possible IP addresses

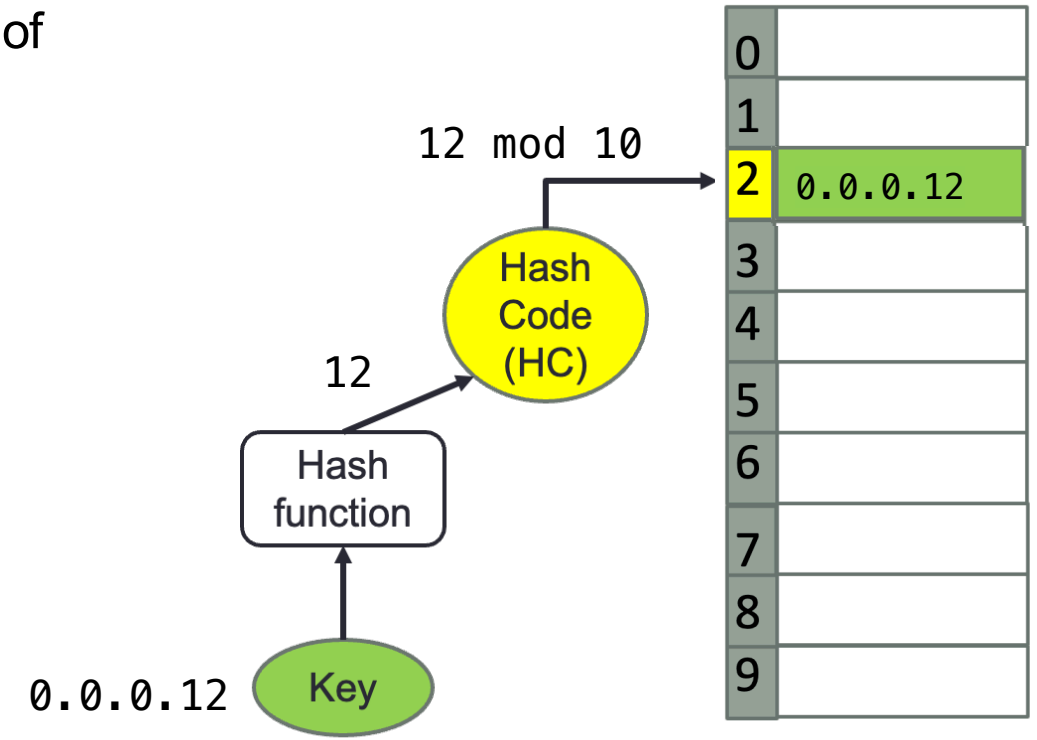
- Keep track of evolving set S whose size is much less than the universe of all possible keys
- For example, 200M unique users (~ 5% of all possible IPs)



Structure of a hash table

- Keys stored in buckets (vector)
- Keys used to compute index of position in vector

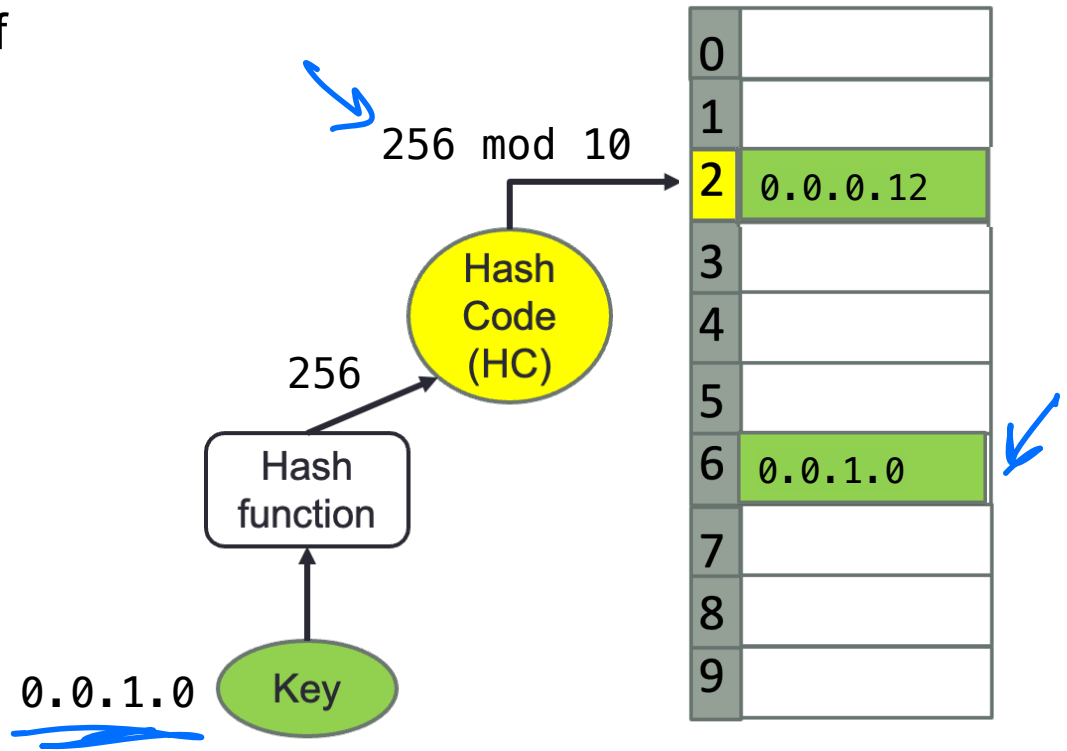
Hash function: h(x)
Table size: m
Num keys: n



Structure of a hash table

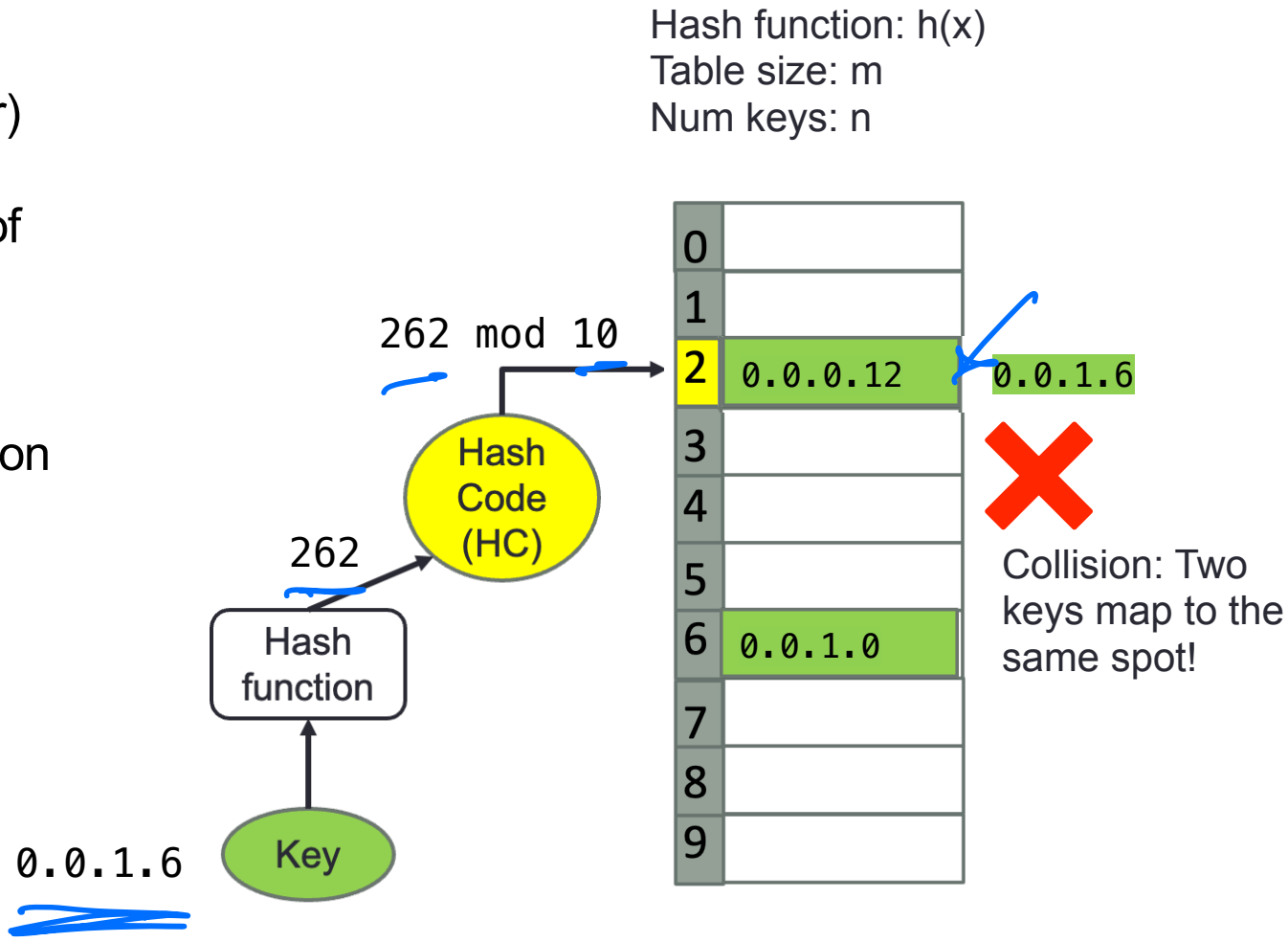
- Keys stored in buckets (vector)
- Keys used to compute index of position in vector

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



Structure of a hash table

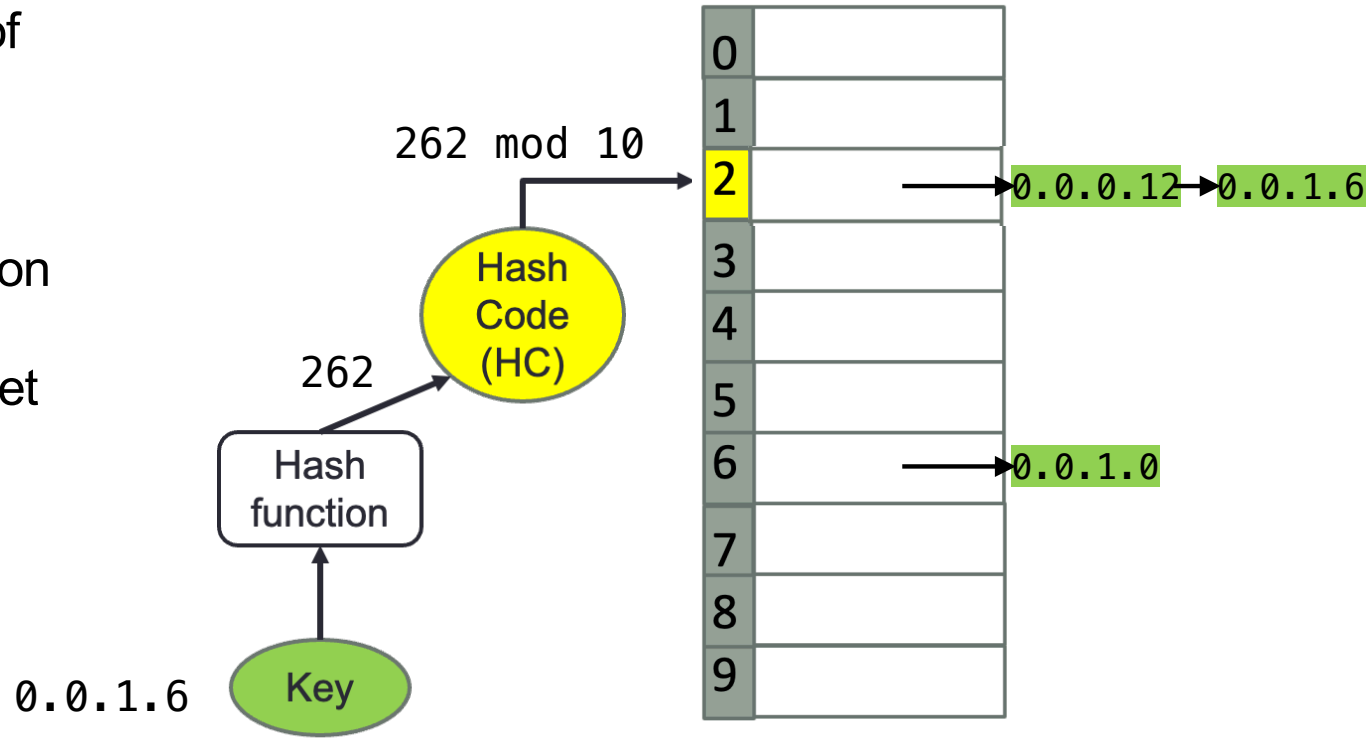
- Keys stored in buckets (vector)
- Keys used to compute index of position in vector
- When two keys map to the same bucket we have a collision



Hash table with separate chaining

- Keys stored in buckets (vector)
- Keys used to compute index of position in vector
- When two keys map to the same bucket we have a collision
- Separate-chaining: each bucket store a list of keys

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



O(1) to search….. Really?

What does O(1) average case really look like?

What's the failure mode?

Code Demo: iptracking.cpp

```cpp
// Hash table with separate chaining
class unordered_set{
 public:
  void insert(Key k) {
      int idx = hash(k) % m;
      buckets[idx].push_back(k);  // O(1) insert at tail
  }

  int count(Key k) {
      int idx = hash(k) % m;
      // Linear search through buckets[idx]
      for (auto& item : buckets[idx]) {
          if (item == k) return 1;
      }
      return 0;
  }

 private:
  vector<list<KeyType>> buckets;  // or vector<forward_list<>>
};
```

Activity 1: Hash Function Practice (5 min)

Task: Given a hash table of size 10 and h(x) = x mod 10:

- Insert keys: 23, 45, 12, 78, 32, 55, 92
- Identify which keys collide
- Count how many buckets are occupied
- What is the load factor α = n/m?

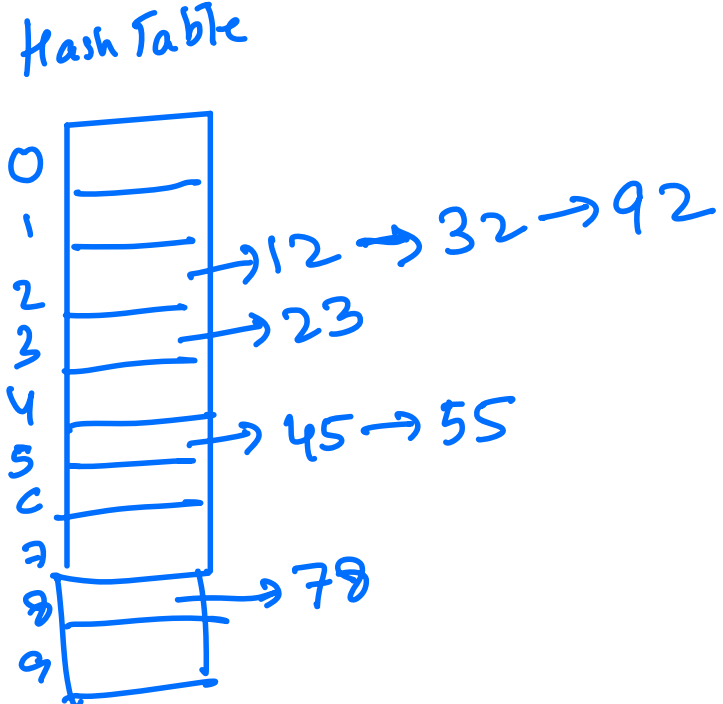
m: no. of buckets

n : no. of keys

m = 10

n = 7

load factor $\alpha = \frac{n}{m} = \frac{7}{10} = 0.7$



Average case analysis

Setup:

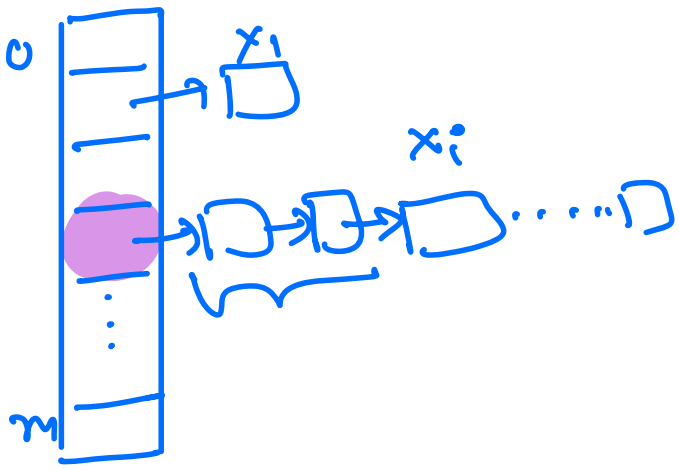
- n keys, m buckets
- Load factor: $\alpha = n/m$
- Assume: uniform hashing (each bucket equally likely)

Derive: Average time to search for a key = ?

Let's assume we inserted keys $x_1, x_2, x_3 \ldots x_i, \ldots x_n$

x_i is the ith key inserted.

Avg. no. of keys in the same bucket as x_i that were inserted before x_i



Assume that keys are uniformly hashed into m buckets.

Avg. size of a bucket = Avg. length of a chain after k inserts $= \frac{k}{m}$

C_i = no. of avg. keys in same bucket as x_i inserted before x_i + 1

$= \left(\frac{i-1}{m}\right) + 1$

C: cost to search for any key

$$E[C] = \sum_{i=1}^{n} \underbrace{Pr(\text{search for key } x_i)}_{\frac{1}{n}} \cdot C_i$$

Assume: equally likely to search for any key

$$= \frac{1}{n} \sum_{i=1}^{n} \left(\frac{i-1}{m} + 1\right)$$

$$= \frac{1}{n} \left(\sum_{i=1}^{n} \frac{i-1}{m} + \sum_{i=1}^{n} 1 \right)$$

$$= \frac{1}{n} \left(\sum_{j=0}^{n-1} \frac{j}{m} + n \right) \quad \text{let } j = i-1$$

$$= \frac{1}{n}\left(\frac{1}{m}\sum_{j=0}^{n-1} j + n\right)$$

$$= \frac{1}{n}\left(\frac{1}{m}\cdot\frac{(n-1)\cdot n}{2} + n\right)$$

$$= \left(\frac{n}{2m} - \frac{1}{2m} + 1\right)$$

$$= 1 + \frac{\alpha}{2} - \frac{1}{2m} \quad \Big| \; \alpha = \frac{n}{m}$$

$$\leq 1 + \frac{\alpha}{2}$$

$$= O\left(1 + \frac{\alpha}{2}\right)$$

iClicker: Determining the hash table size

You want to place 1000 elements in a hash table, using chained hashing.
You'd like a successful search to examine 2 keys on average.
How big does the table need to be?

A. 500

B. 1000

C. 250

D. 125

E. 2000

Avg. no of keys to search for a successful search $= 1 + \frac{\alpha}{2}$

$1 + \frac{\alpha}{2} = 2$

$\frac{\alpha}{2} = 1$

$\frac{n}{2m} = 1$

$\Rightarrow m = \frac{n}{2} = 500$

Two flavors of hash tables in C++ STL

```
#include <unordered_set>  // unique keys only
#include <unordered_map>  // key-value pairs
```

	unordered_set<T>	unordered_map<K,V>
Stores	Keys only	Key-value pairs
Use case	"Have I seen this?"	"What's the value for this key?"
Example	Tracking unique IPs	Counting visits per IP

STL unordered_set

```
unordered_set<string> seen_ips;

// Insert - O(1) average
seen_ips.insert("192.168.1.1");
seen_ips.insert("10.0.0.5");
seen_ips.insert("192.168.1.1");  // ignored, already exists

// Check membership - O(1) average
if (seen_ips.count("192.168.1.1")) {  // returns 0 or 1
    cout << "Seen before!";
}

// Size
cout << seen_ips.size();  // 2 (not 3!)

// Remove - O(1) average
seen_ips.erase("10.0.0.5");
```

STL unordered_map

```
unordered_map<string, int> visit_count;

// Insert / Update - O(1) average
visit_count["192.168.1.1"] = 1;     // insert
visit_count["192.168.1.1"]++;       // update: now 2
visit_count[“10.0.0.5"]++; //auto-creates with 0, then increments to 1

// Lookup - O(1) average
cout << visit_count["192.168.1.1"];  // 2

// Check if key exists (without creating it!)
if (visit_count.count("8.8.8.8")) {
     cout << visit_count["8.8.8.8"];
}
// Iterate all pairs
for (auto& [ip, count] : visit_count) {
     cout << ip << ": " << count << "\n";
 }
```

Web traffic analysis

You're back at your job at Amazon. Your boss hands you a log of IP addresses from today's traffic and asks:

1. How many unique visitors did we have?
2. How many times did each IP visit?
3. Which IP visited the most?
4. Who are our top K most frequent visitors?
5. Flag any suspicious IPs (visited > threshold times)

Open traffic_activity.cpp

Q4 Top K-most frequent visitors

Approach 1: Insert all n keys from unordered_map (ip : count) into the PQ (count : ip)

PQ → max-heap - size n

n inserts into a PQ of size (max n)

$O(n \log n + k \log n)$

pop

Approach 2: Use heapify

① Batch insert n keys into a PQ

priority_queue < pair < int , string > >

PQ (v.begin(), v.end())

heapify O(n)

② Do k-pops — $O(k \log n)$

Overall : $O(n) + O(k \log n)$

Approach 3: Use a min-heap

Explore on your own.

$O(n + k \log k)$

When to use a hash table, when not to?

Mapping Unique Identifiers to Values Hash tables are great for associative arrays, dictionaries, and maps

Fast Key-Value Access When you frequently need to retrieve values by a unique key (e.g., looking up user by ID).

Datasets with Unordered Data If you have a dataset and don't require data to be stored in a particular order. Other ordered structures like BST or linked lists, need O(logn) or O(n) time for lookups

Frequent Insertions and Deletions

Since hash tables handle these operations in average O(1) time, they're ideal for use cases where data is constantly added or removed,

Example: caching, session management, or real-time tracking.

Wrap up

Enjoy the long weekend!

Continue with working on lp 04, lab04

Quiz 3 on Wednesday of next week

Topics include: BST, iterators stacks, queues, and complete binary trees

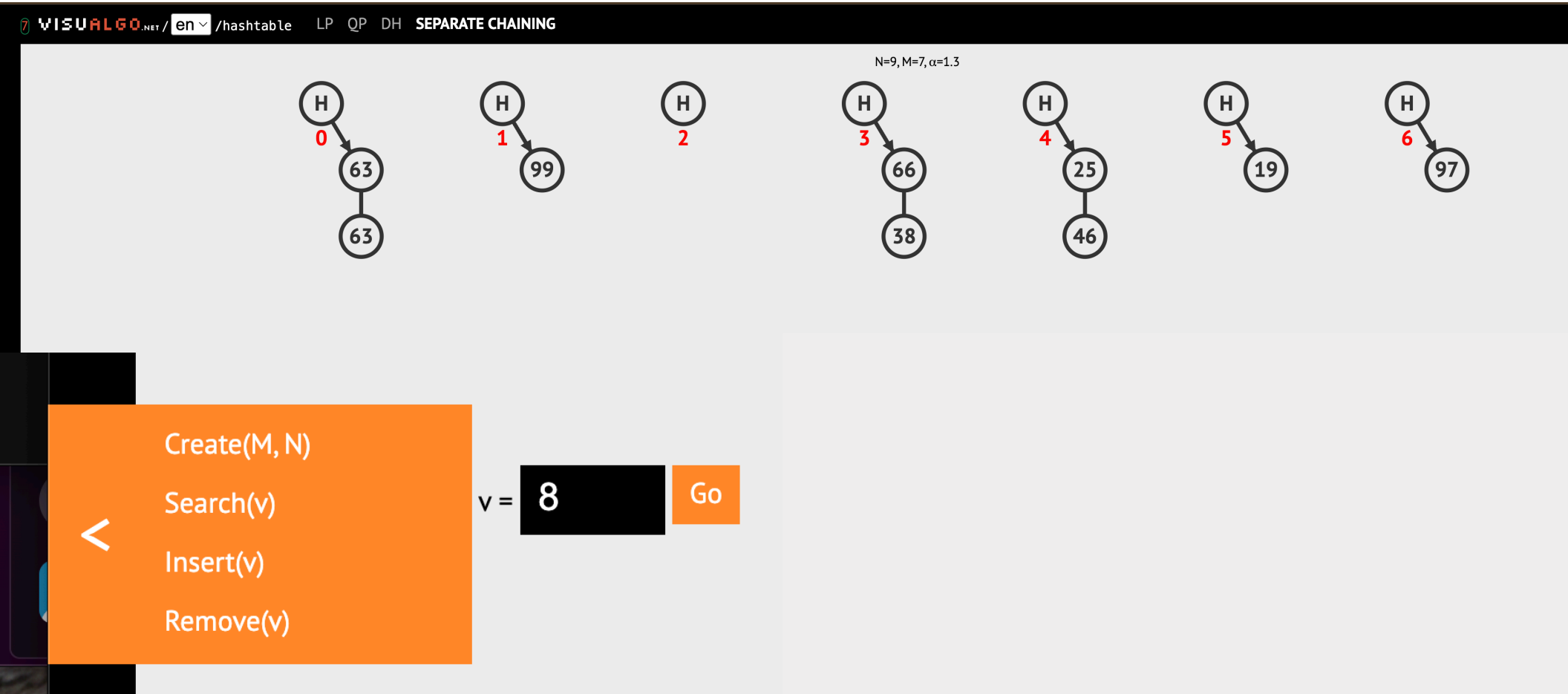
Lectures 5 to 9

Leetcode problem sets 2 and 3

Lab 02 and lab03

Hash table visualization

https://visualgo.net/en/hashtable



References

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

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