Introduction:

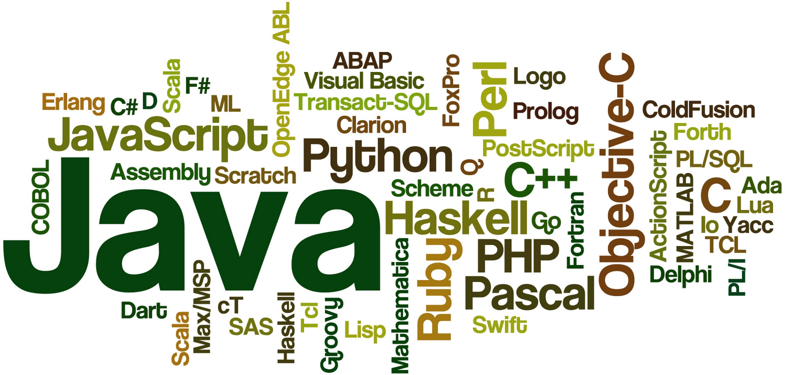
Michael Levin

Hash Tables

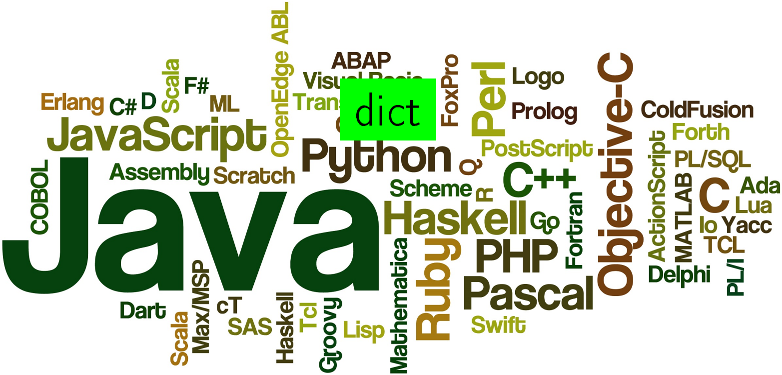
Data Structures and Algorithms
Algorithmic Toolbox

Outline

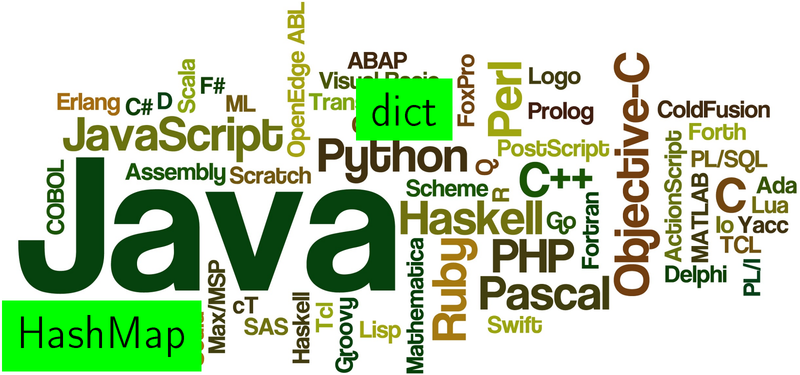
Programming Languages



Programming Languages



Programming Languages



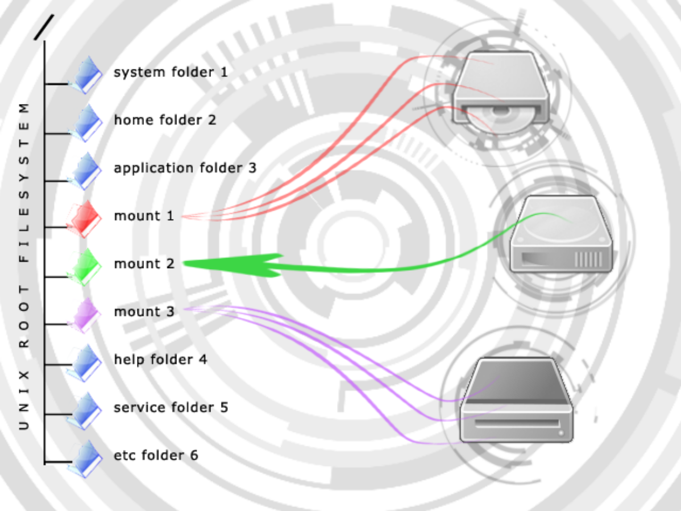
Programming Languages

dict

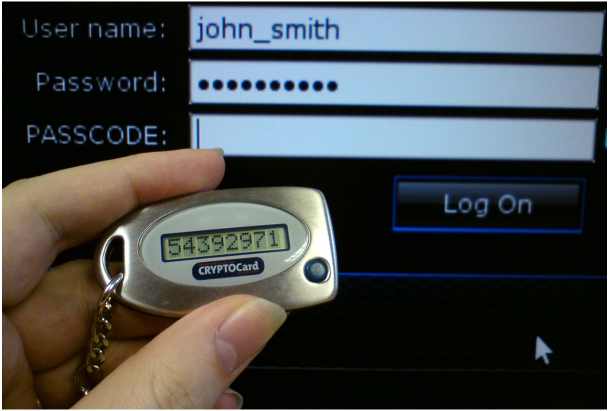
HashMap

for, if, while, int

File Systems



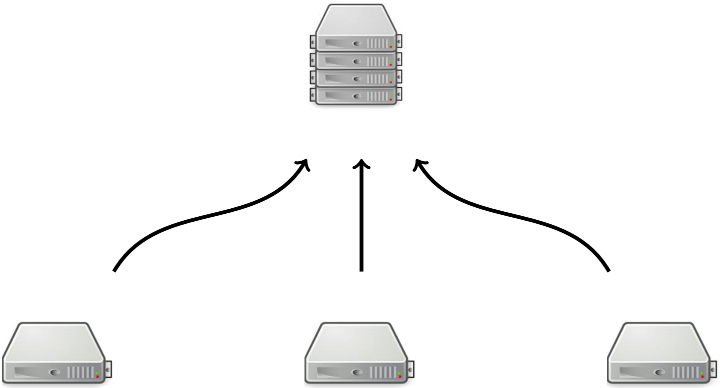
Password Verification



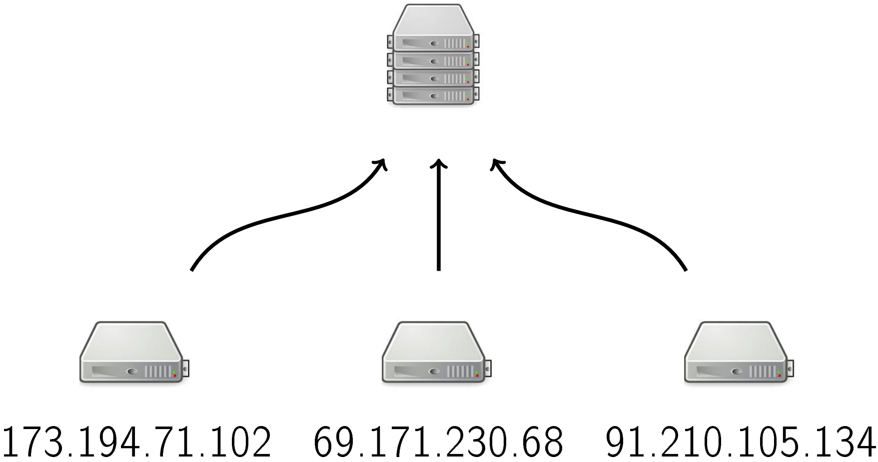
Storage Optimization

Outline

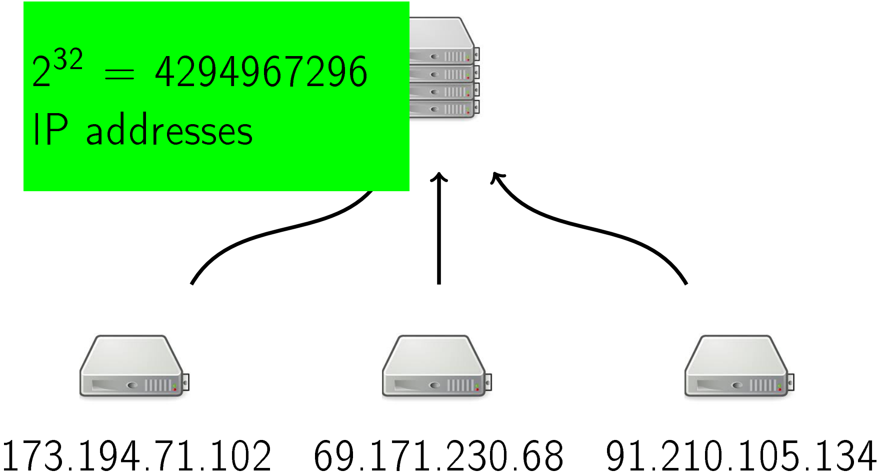
Web Service



Web Service



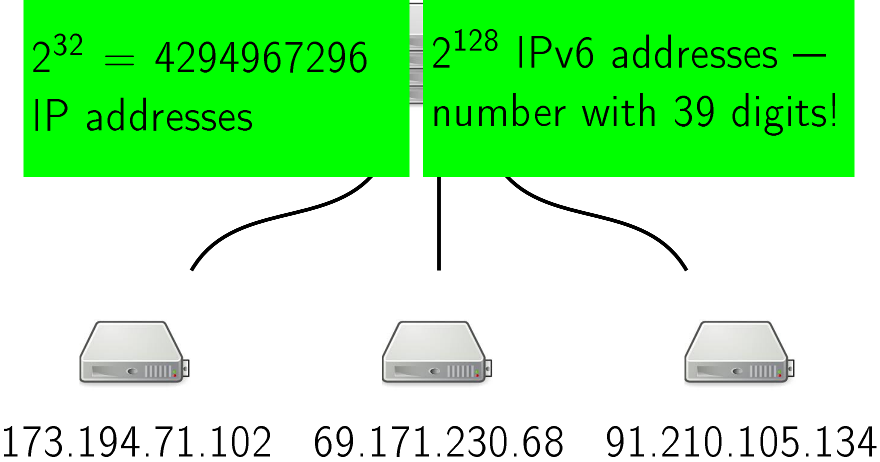
Web Service


$2^{32} = 4294967296$
IP addresses
173.194.71.102
69.171.230.68
91.210.105.134

Web Service

$2^{32} = 4294967296$ IP addresses

2^{128} IPv6 addresses — number with 39 digits!



Access Log

Date	Time	IP address
09 Dec 2015	00:45:13	173.194.71.102
09 Dec 2015	00:45:15	69.171.230.68
...	...	...
...	...	...
09 Dec 2015	01:45:13	91.210.105.134

IP Access List

Analyse the access log and quickly answer queries: did anybody access the service from this *IP* during the last hour? How many times? How many *IP*s were used to access the service during the last hour?

Log Processing

- 1h of logs can contain millions of lines

Log Processing

- 1h of logs can contain millions of lines
- Too slow to process that for each query

Log Processing

- 1h of logs can contain millions of lines
- Too slow to process that for each query
- Keep count: how many times each IP appears in the last 1h of the access log

Log Processing

- 1h of logs can contain millions of lines
- Too slow to process that for each query
- Keep count: how many times each IP appears in the last 1h of the access log
- C is some data structure to store the mapping from IPs to counters

Log Processing

- 1h of logs can contain millions of lines
- Too slow to process that for each query
- Keep count: how many times each IP appears in the last 1h of the access log
- C is some data structure to store the mapping from IPs to counters
- We will learn later how to implement C

Log Processing

Time	IP address
00:45:13	173.194.71.102
00:45:13	69.171.230.68
...	...
01:45:13	173.194.71.102
01:45:13	91.210.105.134

Log Processing

Time	IP address
00:45:13	173.194.71.102
00:45:13	69.171.230.68
...	...
01:45:13	173.194.71.102
01:45:13	91.210.105.134

Now

Log Processing

Time	IP address
00:45:13	173.194.71.102
00:45:13	69.171.230.68
...	...
01:45:13	173.194.71.102
01:45:13	91.210.105.134

Now

Increment counter

Log Processing

Time	IP address
00:45:13	173.194.71.102
00:45:13	69.171.230.68
...	...
01:45:13	173.194.71.102
01:45:13	91.210.105.134

1 hour ago

Now

Increment counter

Log Processing

Decrement counter

1 hour ago

Time	IP address
00:45:13	173.194.71.102
00:45:13	69.171.230.68
...	...
01:45:13	173.194.71.102
01:45:13	91.210.105.134

Now

Increment counter

Main Loop

log - array of log lines $(time, IP)$
C - mapping from IPs to counters
i - first unprocessed log line
j - first line in current 1h window
$i \leftarrow 0$
$j \leftarrow 0$
$C \leftarrow \emptyset$
Each second
  UpdateAccessList(log, i, j, C)

UpdateAccessList(log, i, j, C)

```
while log[i].time ≤ Now():
    C[log[i].IP] ← C[log[i].IP] + 1
    i ← i + 1
while log[j].time ≤ Now() − 3600:
    C[log[j].IP] ← C[log[j].IP] − 1
    j ← j + 1
```

```
UpdateAccessList(log, i, j, C)

while log[i].time ≤ Now():
    C[log[i].IP] ← C[log[i].IP] + 1
    i ← i + 1
while log[j].time ≤ Now() − 3600:
    C[log[j].IP] ← C[log[j].IP] − 1
    j ← j + 1
```

```
AccessedLastHour(IP, C)

return C[IP] > 0
```

Coming Next

How to implement the mapping C?

Outline

UpdateAccessList(log, i, j, C)

```
while log[i].time ≤ Now():
    C[log[i].IP] ← C[log[i].IP] + 1
    i ← i + 1
while log[j].time ≤ Now() − 3600:
    C[log[j].IP] ← C[log[j].IP] − 1
    j ← j + 1
```

AccessedLastHour(IP, C)

```
return C[IP] > 0
```

Direct Addressing

- Need a data structure for C

Direct Addressing

- Need a data structure for C
- There are 2^{32} different IP(v4) addresses

Direct Addressing

- Need a data structure for C
- There are 2^{32} different IP(v4) addresses
- Convert IP to 32-bit integer

Direct Addressing

- Need a data structure for C
- There are 2^{32} different IP(v4) addresses
- Convert IP to 32-bit integer
- Create an integer array A of size 2^{32}

Direct Addressing

- Need a data structure for C
- There are 2^{32} different IP(v4) addresses
- Convert IP to 32-bit integer
- Create an integer array A of size 2^{32}
- Use $A[\mathtt{int}(IP)]$ as $C[IP]$

int(IP)

An IPv4 address (dotted-decimal notation)

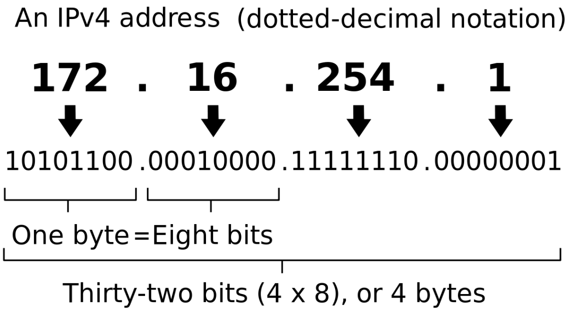
172 . 16 . 254 . 1

10101100 .00010000 .11111110 .00000001

One byte = Eight bits

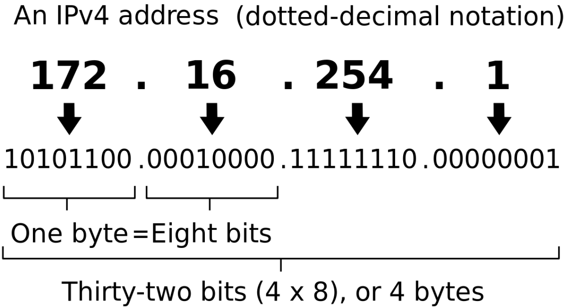
Thirty-two bits (4 x 8), or 4 bytes

int(IP)



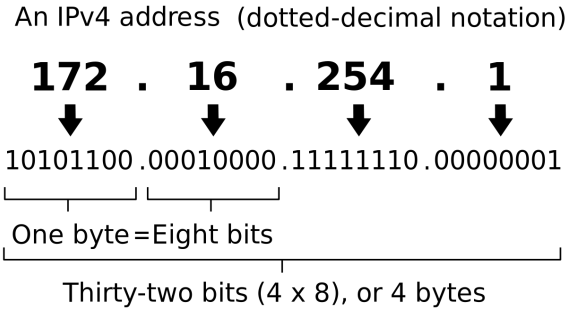
- $\mathtt{int}(0.0.0.1) = 1$

int(IP)



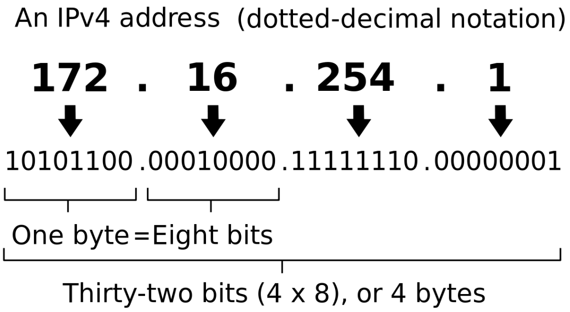
- $\texttt{int}(0.0.0.1) = 1$
- $\texttt{int}(172.16.254.1) = 2886794753$

int(IP)



- $\texttt{int}(0.0.0.1) = 1$
- $\texttt{int}(172.16.254.1) = 2886794753$
- $\texttt{int}(69.171.230.68) =$

int(IP)



- $\mathtt{int}(0.0.0.1) = 1$
- $\mathtt{int}(172.16.254.1) = 2886794753$
- $\mathtt{int}(69.171.230.68) = 1168893508$

int(IP)

`return` $IP[1]\cdot 2^{24} + IP[2]\cdot 2^{16} + IP[3]\cdot 2^{8} + IP[4]$

$\texttt{int}(IP)$

$\texttt{return}\ \ IP[1]\cdot 2^{24}+IP[2]\cdot 2^{16}+IP[3]\cdot 2^{8}+IP[4]$

$\texttt{UpdateAccessList}(log, i, j, A)$

$\texttt{while}\ \ log[i].time \leq Now():$

$\quad A[\texttt{int}(log[i].IP)] \leftarrow A[\texttt{int}(log[i].IP)] + 1$

$\quad i \leftarrow i + 1$

$\texttt{while}\ \ log[j].time \leq Now() - 3600:$

$\quad A[\texttt{int}(log[j].IP)] \leftarrow A[\texttt{int}(log[j].IP)] - 1$

$\quad j \leftarrow j + 1$

$\texttt{AccessedLastHour}(IP)$

$\texttt{return } A[\texttt{int}(IP)] > 0$

Asymptotics

- `UpdateAccessList` is $O(1)$ per log line

Asymptotics

- `UpdateAccessList` is $O(1)$ per log line
- `AccessedLastHour` is $O(1)$

Asymptotics

- `UpdateAccessList` is $O(1)$ per log line
- `AccessedLastHour` is $O(1)$
- But need 2^{32} memory even for few IPs

Asymptotics

- `UpdateAccessList` is $O(1)$ per log line
- `AccessedLastHour` is $O(1)$
- But need 2^{32} memory even for few IPs
- IPv6: 2^{128} won't fit in memory

Asymptotics

- `UpdateAccessList` is $O(1)$ per log line
- `AccessedLastHour` is $O(1)$
- But need 2^{32} memory even for few IPs
- IPv6: 2^{128} won't fit in memory
- In general: $O(N)$ memory, $N = |S|$

Outline

- Direct addressing requires too much memory

- Direct addressing requires too much memory
- Let's store only active IPs

- Direct addressing requires too much memory
- Let's store only active IPs
- Store them in a list

- Direct addressing requires too much memory
- Let's store only active IPs
- Store them in a list
- Store only last occurrence of each IP

- Direct addressing requires too much memory
- Let's store only active IPs
- Store them in a list
- Store only last occurrence of each IP
- Keep the order of occurrence

Access Log

Time	IP address
00:45:13	173.194.71.102
00:45:13	69.171.230.68
01:00:00	69.171.230.68
01:45:13	173.194.71.102
01:45:13	91.210.105.134

Access Log

Time	IP address
00:45:13	173.194.71.102
00:45:13	69.171.230.68
01:00:00	69.171.230.68
01:45:13	173.194.71.102
01:45:13	91.210.105.134

$\boxed{173.194.71.102}$

Access Log

Time	IP address
00:45:13	173.194.71.102
00:45:13	69.171.230.68
01:00:00	69.171.230.68
01:45:13	173.194.71.102
01:45:13	91.210.105.134

173.194.71.102

69.171.230.68

Access Log

Time	IP address
00:45:13	173.194.71.102
00:45:13	69.171.230.68
01:00:00	69.171.230.68
01:45:13	173.194.71.102
01:45:13	91.210.105.134

173.194.71.102

Access Log

Time	IP address
00:45:13	173.194.71.102
00:45:13	69.171.230.68
01:00:00	69.171.230.68
01:45:13	173.194.71.102
01:45:13	91.210.105.134

173.194.71.102

69.171.230.68

Access Log

Time	IP address
00:45:13	173.194.71.102
00:45:13	69.171.230.68
01:00:00	69.171.230.68
01:45:13	173.194.71.102
01:45:13	91.210.105.134

69.171.230.68

Access Log

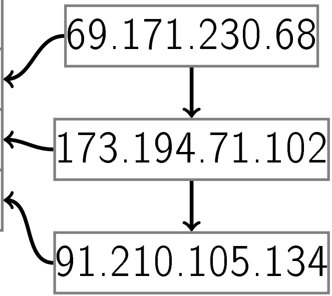
Time	IP address
00:45:13	173.194.71.102
00:45:13	69.171.230.68
01:00:00	69.171.230.68
01:45:13	173.194.71.102
01:45:13	91.210.105.134

$\boxed{69.171.230.68}$

$\boxed{173.194.71.102}$

Access Log

Time	IP address
00:45:13	173.194.71.102
00:45:13	69.171.230.68
01:00:00	69.171.230.68
01:45:13	173.194.71.102
01:45:13	91.210.105.134



UpdateAccessList(log, i, j, L)

```
while log[i].time ≤ Now():
    if L.Find(log[i].IP):
        L.Erase(log[i].IP)
    L.Append(log[i].IP)
    i ← i + 1
while log[j].time ≤ Now() − 3600:
    if L.Top() == log[j].IP:
        L.Pop()
    j ← j + 1
```

```
AccessedLastHour(IP, L)
```

```
return L.Find(IP)
```

Asymptotics

- n is number of active IPs

Asymptotics

- n is number of active IPs
- Memory usage is $\Theta(n)$

Asymptotics

- n is number of active IPs
- Memory usage is $\Theta(n)$
- $L.\texttt{Append}$, $L.\texttt{Top}$, $L.\texttt{Pop}$ are $\Theta(1)$

Asymptotics

- n is number of active IPs
- Memory usage is $\Theta(n)$
- $L.\texttt{Append}$, $L.\texttt{Top}$, $L.\texttt{Pop}$ are $\Theta(1)$
- $L.\texttt{Find}$ and $L.\texttt{Erase}$ are $\Theta(n)$

Asymptotics

- n is number of active IPs
- Memory usage is $\Theta(n)$
- $L.\texttt{Append}$, $L.\texttt{Top}$, $L.\texttt{Pop}$ are $\Theta(1)$
- $L.\texttt{Find}$ and $L.\texttt{Erase}$ are $\Theta(n)$
- `UpdateAccessList` is $\Theta(n)$ per log line

Asymptotics

- n is number of active IPs
- Memory usage is $\Theta(n)$
- L.`Append`, L.`Top`, L.`Pop` are $\Theta(1)$
- L.`Find` and L.`Erase` are $\Theta(n)$
- `UpdateAccessList` is $\Theta(n)$ per log line
- `AccessedLastHour` is $\Theta(n)$

Outline

Encoding IPs

- Encode IPs with small numbers

Encoding IPs

- Encode IPs with small numbers
- I.e. numbers from 0 to 999

Encoding IPs

- Encode IPs with small numbers
- I.e. numbers from 0 to 999
- Different codes for currently active IPs

Hash Function

Definition

For any set of objects S and any integer $m > 0$, a function $h : S \to \{0, 1, \ldots, m-1\}$ is called a hash function.

Hash Function

Definition

For any set of objects S and any integer $m > 0$, a function $h : S \to \{0, 1, \ldots, m-1\}$ is called a hash function.

Definition

m is called the cardinality of hash function h.

Desirable Properties

- h should be fast to compute

Desirable Properties

- h should be fast to compute
- Different values for different objects

Desirable Properties

- h should be fast to compute
- Different values for different objects
- Direct addressing with $O(m)$ memory

Desirable Properties

- h should be fast to compute
- Different values for different objects
- Direct addressing with $O(m)$ memory
- Want small cardinality m

Desirable Properties

- h should be fast to compute
- Different values for different objects
- Direct addressing with $O(m)$ memory
- Want small cardinality m
- Impossible to have all different values if number of objects $|S|$ is more than m

Collisions

Definition

When $h(o_1) = h(o_2)$ and $o_1 \neq o_2$, this is a collision.

Outline

Map

Store mapping from objects to other objects:

- Filename $\rightarrow$ location of the file on disk
- Student ID $\rightarrow$ student name
- Contact name $\rightarrow$ contact phone number

Map

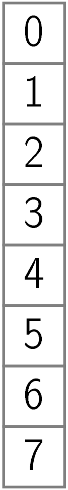
Store mapping from objects to other objects:

- Filename $\to$ location of the file on disk
- Student ID $\to$ student name
- Contact name $\to$ contact phone number

Definition

Map from S to V is a data structure with methods $\texttt{HasKey}(O)$, $\texttt{Get}(O)$, $\texttt{Set}(O, v)$, where $O \in S, v \in V$.

Chaining



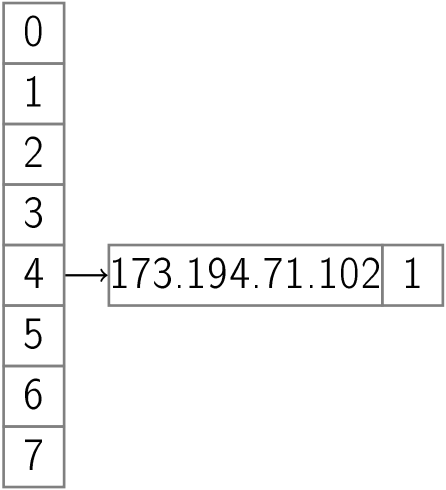
Chaining

$$h(173.194.71.102) = 4$$

0
1
2
3
4
5
6
7

Chaining

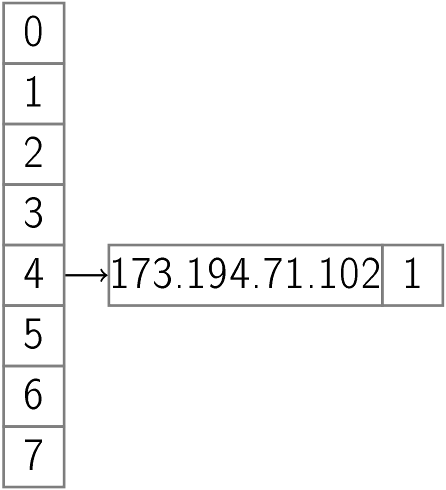
$$h(173.194.71.102) = 4$$



Chaining

$$h(173.194.71.102) = 4$$

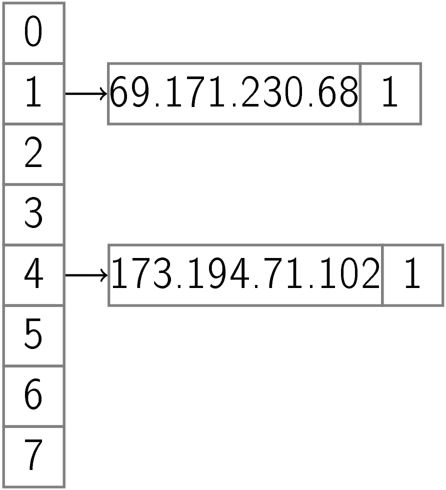
$$h(69.171.230.68) = 1$$



Chaining

$$h(173.194.71.102) = 4$$

$$h(69.171.230.68) = 1$$

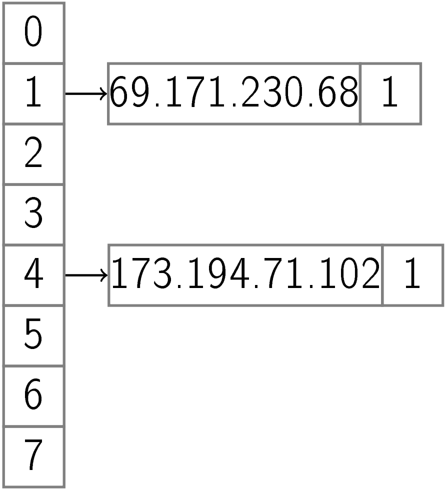


Chaining

$$h(173.194.71.102) = 4$$

$$h(69.171.230.68) = 1$$

$$h(173.194.71.102) = 4$$

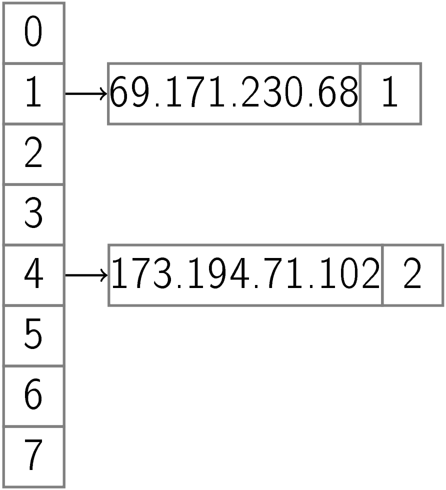


Chaining

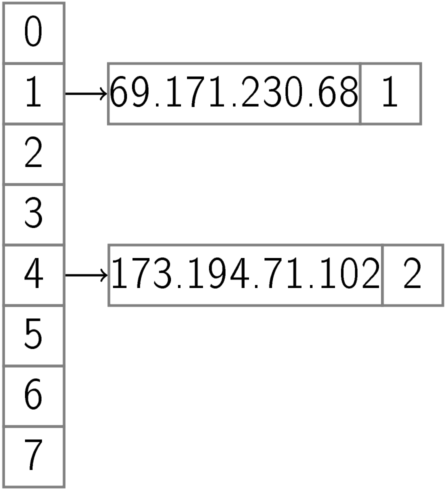
$$h(173.194.71.102) = 4$$

$$h(69.171.230.68) = 1$$

$$h(173.194.71.102) = 4$$



Chaining



$$h(173.194.71.102) = 4$$

$$h(69.171.230.68) = 1$$

$$h(173.194.71.102) = 4$$

$$h(91.210.105.134) = 4$$

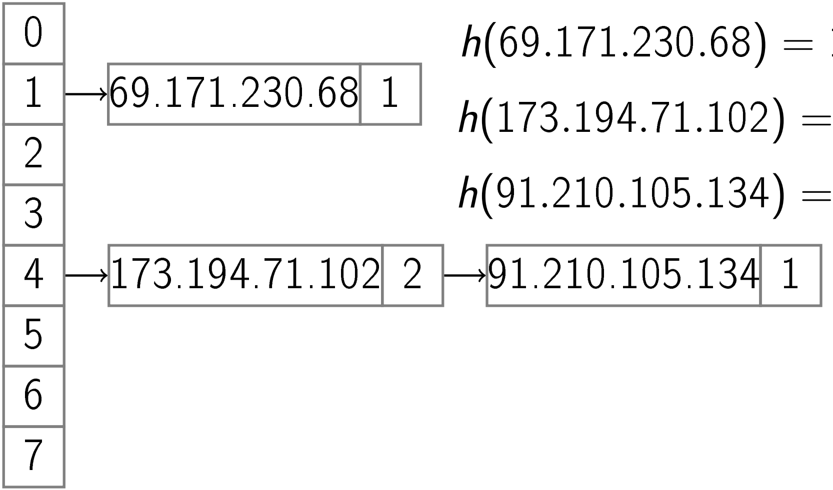
Chaining

$$h(173.194.71.102) = 4$$

$$h(69.171.230.68) = 1$$

$$h(173.194.71.102) = 4$$

$$h(91.210.105.134) = 4$$



$h : S \to \{0, 1, \ldots, m-1\}$

$O, O' \in S$

$v, v' \in V$

$A \leftarrow$ array of m lists (chains) of pairs (O, v)

`HasKey`(O)

$L \leftarrow A[h(O)]$

for (O', v') in L:

  if $O' == O$:

    return true

return false

```
Get(O)

L ← A[h(O)]
for (O′, v′) in L:
  if O′ == O:
    return v′
return n/a
```

Set(O, v)

```
L ← A[h(O)]
for p in L:
    if p.O == O:
        p.v ← v
        return
L.Append(O, v)
```

Lemma

Let c be the length of the longest chain in A. Then the running time of `HasKey`, `Get`, `Set` is $\Theta(c+1)$.

Lemma

Let c be the length of the longest chain in A. Then the running time of `HasKey`, `Get`, `Set` is $\Theta(c + 1)$.

Proof

- If $L = A[h(O)]$, $\texttt{len}(L) = c$, $O \notin L$, need to scan all c items

Lemma

Let c be the length of the longest chain in A. Then the running time of `HasKey`, `Get`, `Set` is $\Theta(c+1)$.

Proof

- If $L = A[h(O)]$, $\texttt{len}(L) = c$, $O \notin L$, need to scan all c items
- If $c = 0$, we still need $O(1)$ time $\square$

Lemma

Let n be the number of different keys O currently in the map and m be the cardinality of the hash function. Then the memory consumption for chaining is $\Theta(n + m)$.

Lemma

Let n be the number of different keys O currently in the map and m be the cardinality of the hash function. Then the memory consumption for chaining is $\Theta(n + m)$.

Proof

- $\Theta(n)$ to store n pairs (O, v)

Lemma

Let n be the number of different keys O currently in the map and m be the cardinality of the hash function. Then the memory consumption for chaining is $\Theta(n + m)$.

Proof

- $\Theta(n)$ to store n pairs (O, v)
- $\Theta(m)$ to store array A of size m $\square$

Outline

Set

Definition

Set is a data structure with methods $\texttt{Add}(O)$, $\texttt{Remove}(O)$, $\texttt{Find}(O)$.

Set

Definition

Set is a data structure with methods `Add`(O), `Remove`(O), `Find`(O).

Examples

- IPs accessed during last hour

Set

Definition

Set is a data structure with methods `Add`(O), `Remove`(O), `Find`(O).

Examples

- IPs accessed during last hour
- Students on campus

Set

Definition

Set is a data structure with methods `Add`(O), `Remove`(O), `Find`(O).

Examples

- IPs accessed during last hour
- Students on campus
- Keywords in a programming language

Implementing Set

Two ways to implement a set using chaining:

- Set is equivalent to map from S to $V = \{\mathit{true}, \mathit{false}\}$

Implementing Set

Two ways to implement a set using chaining:

- Set is equivalent to map from S to $V = \{true, false\}$
- Store just objects O instead of pairs (O, v) in chains

$h : S \to \{0, 1, \ldots, m-1\}$

$O, O' \in S$

$A \leftarrow$ array of m lists (chains) of objects O

Find(O)

$L \leftarrow A[h(O)]$

```
for O' in L:
  if O' == O:
    return true
return false
```

Add(O)

```
L ← A[h(O)]
for O' in L:
    if O' == O:
        return
L.Append(O)
```

Remove(O)

```
if not Find(O):
  return
L ← A[h(O)]
L.Erase(O)
```

Hash Table

Definition

An implementation of a set or a map using hashing is called a hash table.

Programming Languages

Set:

- `unordered_set` in C++
- `HashSet` in Java
- `set` in Python

Map:

- `unordered_map` in C++
- `HashMap` in Java
- `dict` in Python

Conclusion

- Chaining is a technique to implement a hash table

Conclusion

- Chaining is a technique to implement a hash table
- Memory consumption is $O(n + m)$

Conclusion

- Chaining is a technique to implement a hash table
- Memory consumption is $O(n + m)$
- Operations work in time $O(c + 1)$

Conclusion

- Chaining is a technique to implement a hash table
- Memory consumption is $O(n + m)$
- Operations work in time $O(c + 1)$
- How to make both m and c small?