Hashing: Introduction

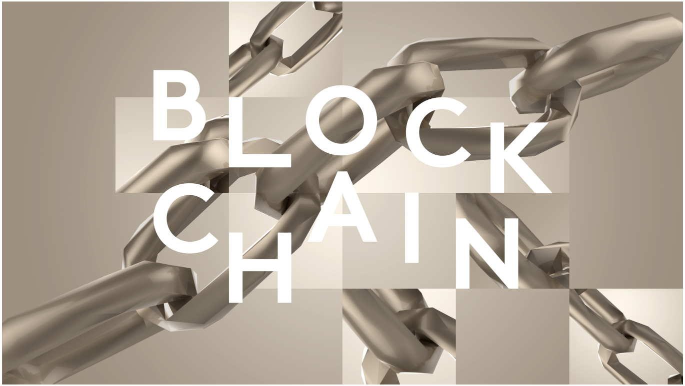
Michael Levin

Department of Computer Science and Engineering
University of California, San Diego

Data Structures Fundamentals
Algorithms and Data Structures

Outline

Blockchain



Programming Languages

Java

Ruby

C#

php

Objective-C

C++

python

Perl

GO

JavaScript

THE C PROGRAMMING LANGUAGE

Visual Basic

Programming Languages

Java

Ruby

C#

php

Objective-C

C++

python

Perl

dict

GO

JavaScript

THE C PROGRAMMING LANGUAGE

Visual Basic

Programming Languages

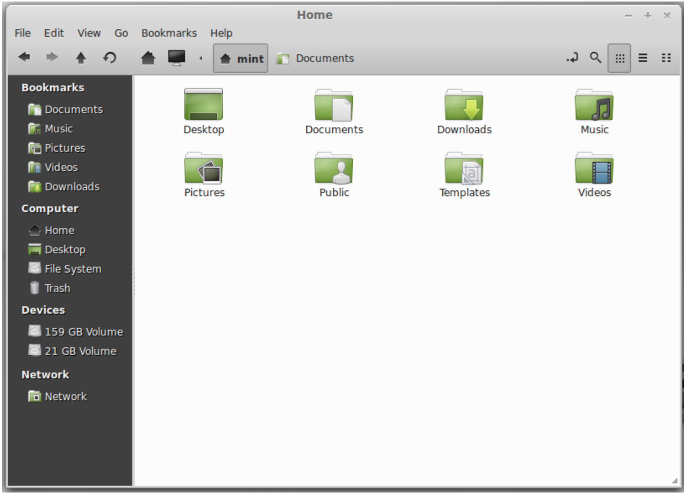
HashMap

dict

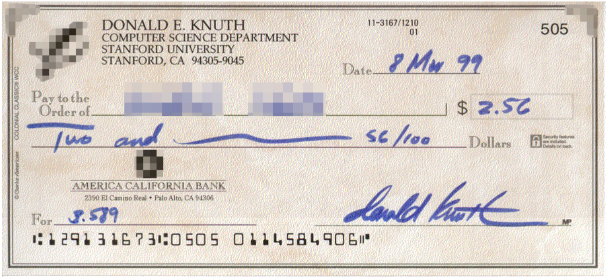
Programming Languages

Keywords: for, if, while, int, ...

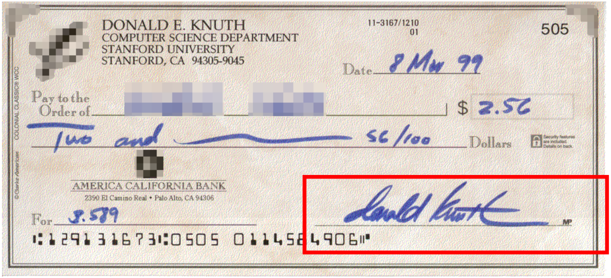
File Systems



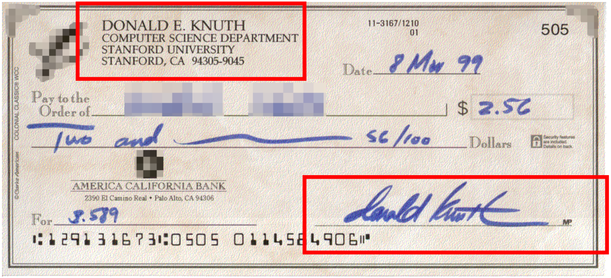
Digital Signature



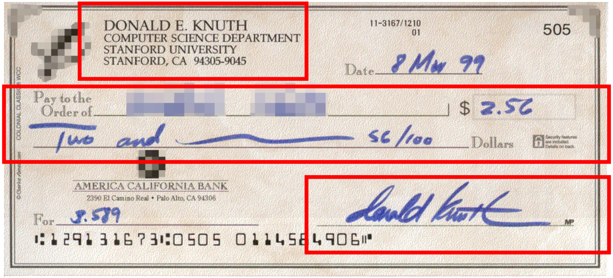
Digital Signature



Digital Signature

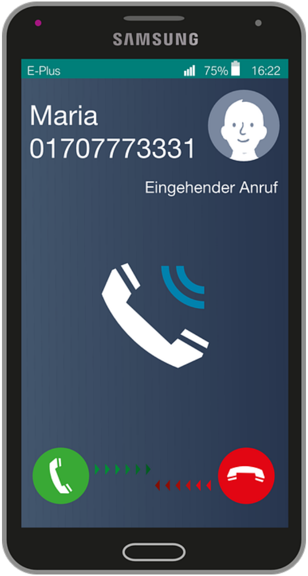


Digital Signature

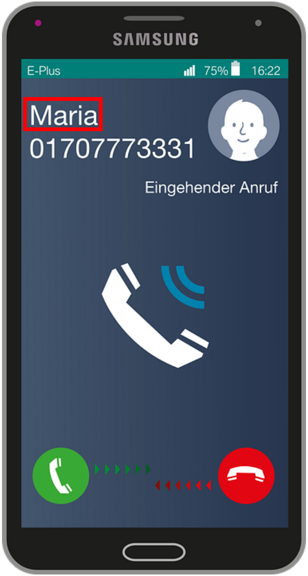


Outline

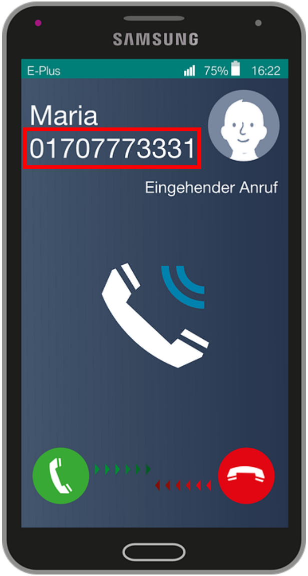
Who's Calling?



Who's Calling?



Who's Calling?



Phone Book

Phone number	Name
01707773331	Maria
239-17-17	Sasha
575-75-75	Helen

Phone to Name

We are going to focus on retrieving name by phone number for now

Local Phone Numbers

- Like 123-23-23

Local Phone Numbers

- Like 123-23-23
- Typically up to 7 digits

Local Phone Numbers

- Like 123-23-23
- Typically up to 7 digits
- Sufficient for $10^7 = 10\,000\,000$ phone numbers

Convert Phone Number to Integer

Examples

123-23-23 $\rightarrow$ 1 232 323

049 12 12 $\rightarrow$ 491 212

5757575 $\rightarrow$ 5 757 575

Direct Addressing

10^7 rows

Phone number	Name
0000000	
...	
2391717	Sasha
...	
5757575	Helen
...	
9999999	

Direct Addressing

- Store phone book as array of size 10^7
- Names are values of the array
- To retrieve name by phone number, convert phone number to integer first
- Use the resulting integer as index in the array of names

```
GetName(phoneNumber)

index ← ConvertToInt(phoneNumber)
return phoneBookArray[index]
```

```
SetName(phoneNumber, name)

index ← ConvertToInt(phoneNumber)
phoneBookArray[index] ← name
```

```
GetName(phoneNumber)

index ← ConvertToInt(phoneNumber)
return phoneBookArray[index]
```

```
SetName(phoneNumber, name)

index ← ConvertToInt(phoneNumber)
phoneBookArray[index] ← name
```

```
GetName(phoneNumber)

index ← ConvertToInt(phoneNumber)
return phoneBookArray[index]
```

```
SetName(phoneNumber, name)

index ← ConvertToInt(phoneNumber)
phoneBookArray[index] ← name
```

GetName(phoneNumber)

```
index ← ConvertToInt(phoneNumber)
return phoneBookArray[index]
```

SetName(phoneNumber, name)

```
index ← ConvertToInt(phoneNumber)
phoneBookArray[index] ← name
```

GetName(phoneNumber)

```
index ← ConvertToInt(phoneNumber)
return phoneBookArray[index]
```

SetName(phoneNumber, name)

```
index ← ConvertToInt(phoneNumber)
phoneBookArray[index] ← name
```

```
GetName(phoneNumber)

index ← ConvertToInt(phoneNumber)
return phoneBookArray[index]
```

```
SetName(phoneNumber, name)

index ← ConvertToInt(phoneNumber)
phoneBookArray[index] ← name
```

Asymptotics

For a phone book with n contacts,

- Retrieve name by phone number in $O(1)$

Asymptotics

For a phone book with n contacts,

- Retrieve name by phone number in $O(1)$
- Set name for a phone number in $O(1)$

Asymptotics

For a phone book with n contacts,

- Retrieve name by phone number in $O(1)$
- Set name for a phone number in $O(1)$
- Memory consumption is $O(|U|)$, where U is the set of all possible phone numbers

Conclusion

- Local phone numbers are up to 7 digits long
- Can store them in an array of size 10^7
- This scheme is called direct addressing
- It is the simplest form of hashing

Outline

International Phone Numbers

- Like $+$1-800-700-00-00

International Phone Numbers

- Like +1-800-700-00-00
- Can be up to 15 digits:
  +594 700 123 233 455

International Phone Numbers

- Like $+1$-800-700-00-00
- Can be up to 15 digits:
  $+594$ 700 123 233 455
- Using direct addressing requires array of size 10^{15}, which would take 7PB (7 petabytes) to store one phone book (1PB $=$ 1024TB, 1TB $=$ 1024GB)

International Phone Numbers

- Like $+1$-800-700-00-00
- Can be up to 15 digits: $+594$ 700 123 233 455
- Using direct addressing requires array of size 10^{15}, which would take 7PB (7 petabytes) to store one phone book ($1\text{PB} = 1024\text{TB}$, $1\text{TB} = 1024\text{GB}$)
- Your phone memory is probably at most 256GB, so you would need 28762 phones to store your phone book :)

Idea

- Direct addressing requires too much memory

Idea

- Direct addressing requires too much memory
- Array is huge because it has a cell for every possible phone number

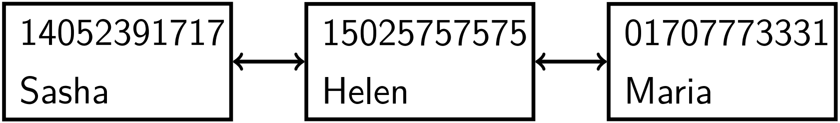
Idea

- Direct addressing requires too much memory
- Array is huge because it has a cell for every possible phone number
- Let's store only the known phone numbers

Idea

- Direct addressing requires too much memory
- Array is huge because it has a cell for every possible phone number
- Let's store only the known phone numbers
- Put pairs (Phone number, Name) into a doubly-linked list

Idea



Operations

- To add a contact, just insert new pair (Phone number, Name) into the list in $O(1)$

Operations

- To add a contact, just insert new pair (Phone number, Name) into the list in $O(1)$
- To retrieve name by phone number, search through the list...

Operations

- To add a contact, just insert new pair (Phone number, Name) into the list in $O(1)$
- To retrieve name by phone number, search through the list...
- ...in $O(n)$, where n is the total number of contacts

Operations

- To add a contact, just insert new pair (Phone number, Name) into the list in $O(1)$
- To retrieve name by phone number, search through the list...
- ...in $O(n)$, where n is the total number of contacts
- Too slow

Idea 2

- Retrieving a name by phone number is slow, because we need to look through the whole list

Idea 2

- Retrieving a name by phone number is slow, because we need to look through the whole list
- Let's put the pairs (Phone number, Name) in a dynamic array sorted by phone number!

Idea 2

01707773331	Maria
14052391717	Sasha
15025757575	Helen

Operations

- Retrieve name by phone number using binary search in $O(\log n)$

Operations

- Retrieve name by phone number using binary search in $O(\log n)$
- To insert a new contact, find appropriate position in $O(\log n)$, then insert in...

Operations

- Retrieve name by phone number using binary search in $O(\log n)$
- To insert a new contact, find appropriate position in $O(\log n)$, then insert in...
- ...$O(n)$, because we need to first move part of the array 1 position to the right

Operations

- Retrieve name by phone number using binary search in $O(\log n)$
- To insert a new contact, find appropriate position in $O(\log n)$, then insert in...
- ...$O(n)$, because we need to first move part of the array 1 position to the right
- Too slow again

Conclusion

- International numbers can be up to 15 digits long
- Direct addressing requires 7 petabytes of memory
- Simple list-based and array-based approaches are too slow
- Next videos — solution using hashing

Outline

Encoding Phone Numbers

- Encode international phone numbers with small numbers

Encoding Phone Numbers

- Encode international phone numbers with small numbers
- E.g. numbers from 0 to 999

Encoding Phone Numbers

- Encode international phone numbers with small numbers
- E.g. numbers from 0 to 999
- Different codes for the phone numbers in the phone book

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

- Hash function should be fast to compute

Desirable Properties

- Hash function should be fast to compute
- Different values for different objects

Desirable Properties

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

Desirable Properties

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

Desirable Properties

- Hash function 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 (by pigeonhole principle)

Collisions

Definition

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

Desirable Properties

- Hash function should be fast to compute

Desirable Properties

- Hash function should be fast to compute
- ~~Different values for different objects~~ Small probability of collision

Desirable Properties

- Hash function should be fast to compute
- ~~Different values for different objects~~ Small probability of collision
- Small enough cardinality m

Outline

Map

Store mapping from objects to other objects:

- Filename $\rightarrow$ location of the file
- Phone number $\rightarrow$ name
- Name $\rightarrow$ phone number

Map

Definition

Map from set S of `objects` to set V of `values` is a data structure with methods `HasKey(object)`, `Get(object)`, `Set(object, value)`, where `object` $\in S$, `value` $\in V$.

Map

Definition

In a Map from S to V, `objects` from S are usually called keys of the Map. Objects from V are called values of the Map.

Chaining for Phone Book

0
1
2
3
4
5
6
7

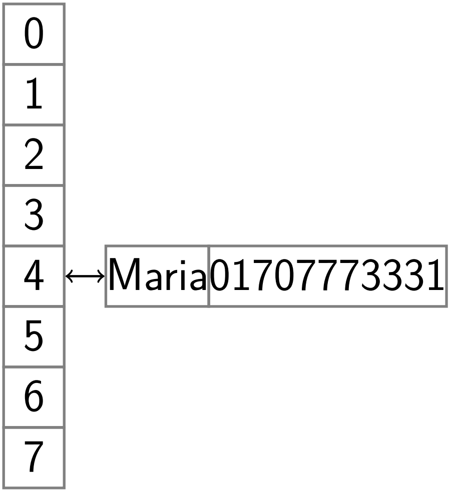
Chaining for Phone Book

$$h(01707773331) = 4$$

0
1
2
3
4
5
6
7

Chaining for Phone Book

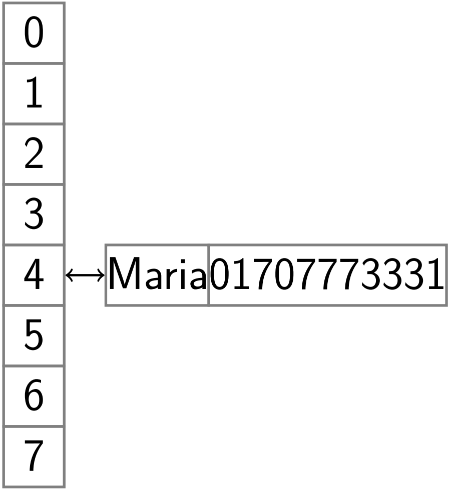
$$h(01707773331) = 4$$



Chaining for Phone Book

$$h(01707773331) = 4$$

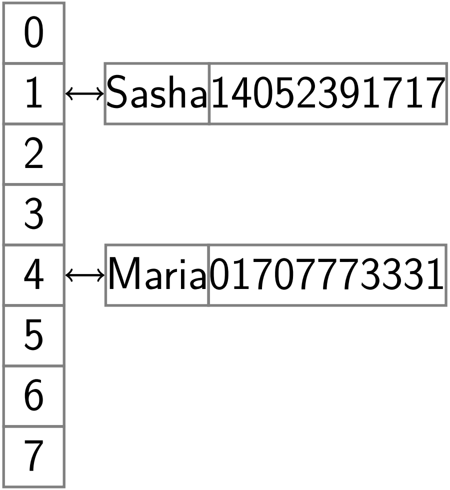
$$h(14052391717) = 1$$



Chaining for Phone Book

$$h(01707773331) = 4$$

$$h(14052391717) = 1$$

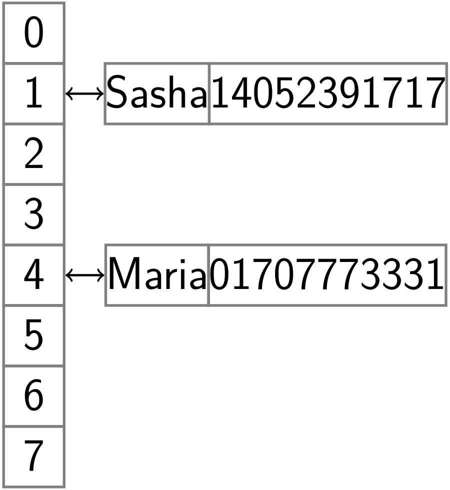


Chaining for Phone Book

$$h(01707773331) = 4$$

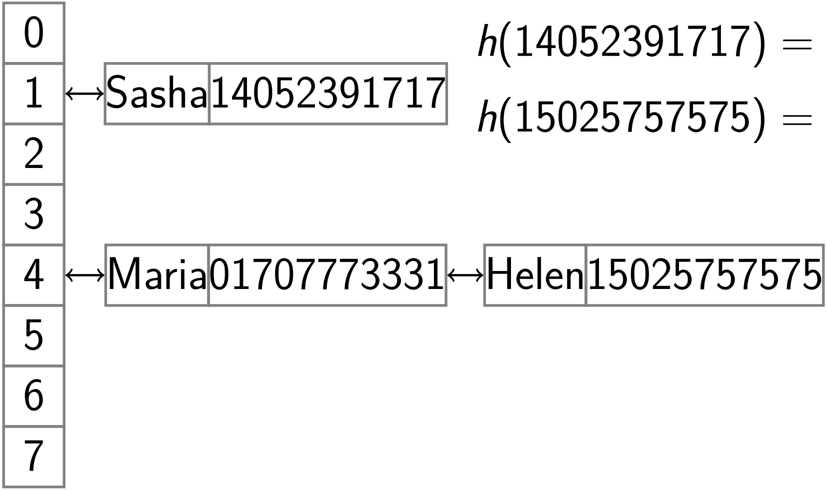
$$h(14052391717) = 1$$

$$h(15025757575) = 4$$



Chaining for Phone Book

$$h(01707773331) = 4$$
$$h(14052391717) = 1$$
$$h(15025757575) = 4$$



Chaining for Phone Book

- Select hash function h of cardinality m

Chaining for Phone Book

- Select hash function h of cardinality m
- Create array `Chains` of size m

Chaining for Phone Book

- Select hash function h of cardinality m
- Create array `Chains` of size m
- Each element of `Chains` is a doubly-linked list of pairs (`name`, `phoneNumber`), called *chain*

Chaining for Phone Book

- Select hash function h of cardinality m
- Create array `Chains` of size m
- Each element of `Chains` is a doubly-linked list of pairs (`name`, `phoneNumber`), called *chain*
- Pair (`name`, `phoneNumber`) goes into chain at position h(`ConvertToInt(phoneNumber)`) in the array `Chains`

Chaining for Phone Book

- To look up name by phone number, go to the chain corresponding to phone number and look through all pairs

Chaining for Phone Book

- To look up name by phone number, go to the chain corresponding to phone number and look through all pairs
- To add a contact, create a pair ($\texttt{name}, \texttt{phoneNumber}$) and insert it into the corresponding chain

Chaining for Phone Book

- To look up name by phone number, go to the chain corresponding to phone number and look through all pairs
- To add a contact, create a pair $(\texttt{name}, \texttt{phoneNumber})$ and insert it into the corresponding chain
- To remove a contact, go to the corresponding chain, find the pair $(\texttt{name}, \texttt{phoneNumber})$ and remove it from the chain

Outline

1. Applications
2. Phone Book
3. International Phone Numbers
4. Hash Functions
5. Chaining
6. **Chaining Implementation and Analysis**
7. Hash Tables

Implementation

`Chains` — array of `chains`

Each `chain` is a list of pairs (object, value)

HasKey(object)

```
chain ← Chains[hash(object)]
for (key, value) in chain:
  if key == object:
    return true
return false
```

Implementation

`Chains` — array of `chains`

Each `chain` is a list of pairs (object, value)

```
HasKey(object)

chain ← Chains[hash(object)]
for (key, value) in chain:
  if key == object:
    return true
return false
```

Implementation

`Chains` — array of chains

Each `chain` is a list of pairs (object, value)

HasKey(object)

```
chain ← Chains[hash(object)]
for (key, value) in chain:
  if key == object:
    return true
return false
```

Implementation

Chains — array of chains

Each chain is a list of pairs (object, value)

HasKey(object)

```
chain ← Chains[hash(object)]
for (key, value) in chain:
  if key == object:
    return true
return false
```

Implementation

`Chains` — array of `chains`

Each `chain` is a list of pairs (object, value)

HasKey(object)

```
chain ← Chains[hash(object)]
for (key, value) in chain:
  if key == object:
    return true
return false
```

Implementation

Chains — array of chains

Each chain is a list of pairs (object, value)

HasKey(object)

```
chain ← Chains[hash(object)]
for (key, value) in chain:
  if key == object:
    return true
return false
```

Implementation

Get(object)

```
chain ← Chains[hash(object)]
for (key, value) in chain:
  if key == object:
    return value
return N/A
```

Implementation

Get(object)

```
chain ← Chains[hash(object)]
for (key, value) in chain:
  if key == object:
    return value
return N/A
```

Implementation

Get(object)

```
chain ← Chains[hash(object)]
for (key, value) in chain:
  if key == object:
    return value
return N/A
```

Implementation

Get(object)

```
chain ← Chains[hash(object)]
for (key, value) in chain:
  if key == object:
    return value
return N/A
```

Implementation

Get(object)

```
chain ← Chains[hash(object)]
for (key, value) in chain:
  if key == object:
    return value
return N/A
```

Implementation

Get(object)

```
chain ← Chains[hash(object)]
for (key, value) in chain:
  if key == object:
    return value
return N/A
```

Implementation

Set(object, value)

```
chain ← Chains[hash(object)]
for pair in chain:
  if pair.key == object:
    pair.value ← value
    return
chain.Append((object, value))
```

Implementation

Set(object, value)

```
chain ← Chains[hash(object)]
for pair in chain:
  if pair.key == object:
    pair.value ← value
    return
chain.Append((object, value))
```

Implementation

Set(object, value)

```
chain ← Chains[hash(object)]
for pair in chain:
  if pair.key == object:
    pair.value ← value
    return
chain.Append((object, value))
```

Implementation

Set(object, value)

```
chain ← Chains[hash(object)]
for pair in chain:
  if pair.key == object:
    pair.value ← value
    return
chain.Append((object, value))
```

Implementation

Set(object, value)

```
chain ← Chains[hash(object)]
for pair in chain:
  if pair.key == object:
    pair.value ← value
    return
chain.Append((object, value))
```

Implementation

Set(object, value)

```
chain ← Chains[hash(object)]
for pair in chain:
  if pair.key == object:
    pair.value ← value
    return
chain.Append((object, value))
```

Implementation

Set(object, value)

```
chain ← Chains[hash(object)]
for pair in chain:
  if pair.key == object:
    pair.value ← value
    return
chain.Append((object, value))
```

Asymptotics

Lemma

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

Asymptotics

Proof

- If the `chain` corresponding to the `object` is non-empty, but the `object` is not found in the `chain`, we will scan all c items — $\Theta(c) = \Theta(c+1)$

Asymptotics

Proof

- If the `chain` corresponding to the `object` is non-empty, but the `object` is not found in the `chain`, we will scan all c items — $\Theta(c) = \Theta(c+1)$
- If $c = 0$, we still need $O(1)$ time, thus the need for "$+1$" $\square$

Asymptotics

Lemma

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

Asymptotics

Proof

- $\Theta(n)$ to store n pairs $(\texttt{object}, \texttt{value})$

Asymptotics

Proof

- $\Theta(n)$ to store n pairs $(\texttt{object}, \texttt{value})$
- $\Theta(m)$ for array `Chains` of size m $\square$

Outline

Set

Definition

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

Set

Examples

- Students on campus

Set

Examples

- Students on campus
- Phone numbers of contacts

Set

Examples

- Students on campus
- Phone numbers of contacts
- Keywords in a programming language

Implementing Set

Two ways to implement a set using chaining:

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

Implementing Set

Two ways to implement a set using chaining:

- Set is equivalent to map from S to $V = \{true\}$
- Store just `objects` instead of pairs (`object`, `value`) in the `chains`

Implementation

Find(object)

```
chain ← Chains[hash(object)]
for key in chain:
  if key == object:
    return true
return false
```

Implementation

Find(object)

```
chain ← Chains[hash(object)]
for key in chain:
  if key == object:
    return true
return false
```

Implementation

Find(object)

```
chain ← Chains[hash(object)]
for key in chain:
  if key == object:
    return true
return false
```

Implementation

Find(object)

```
chain ← Chains[hash(object)]
for key in chain:
    if key == object:
        return true
return false
```

Implementation

Find(object)

```
chain ← Chains[hash(object)]
for key in chain:
  if key == object:
    return true
return false
```

Implementation

Find(object)

```
chain ← Chains[hash(object)]
for key in chain:
  if key == object:
    return true
return false
```

Implementation

Add(object)

```
chain ← Chains[hash(object)]
for key in chain:
  if key == object:
    return
chain.Append(object)
```

Implementation

Add(object)

```
chain ← Chains[hash(object)]
for key in chain:
    if key == object:
        return
chain.Append(object)
```

Implementation

Add(object)

```
chain ← Chains[hash(object)]
for key in chain:
  if key == object:
    return
chain.Append(object)
```

Implementation

Add(object)

```
chain ← Chains[hash(object)]
for key in chain:
  if key == object:
    return
chain.Append(object)
```

Implementation

Add(object)

```
chain ← Chains[hash(object)]
for key in chain:
  if key == object:
    return
chain.Append(object)
```

Implementation

Add(object)

```
chain ← Chains[hash(object)]
for key in chain:
  if key == object:
    return
chain.Append(object)
```

Implementation

Remove(object)

```
if not Find(object):
  return
chain ← Chains[hash(object)]
chain.Erase(object)
```

Implementation

```
Remove(object)

if not Find(object):
  return
chain ← Chains[hash(object)]
chain.Erase(object)
```

Implementation

Remove(object)

```
if not Find(object):
  return
chain ← Chains[hash(object)]
chain.Erase(object)
```

Implementation

Remove(object)

```
if not Find(object):
  return
chain ← Chains[hash(object)]
chain.Erase(object)
```

Implementation

Remove(object)

```
if not Find(object):
  return
chain ← Chains[hash(object)]
chain.Erase(object)
```

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
- Number of objects n, hash function cardinality m, longest chain length c

Conclusion

- Chaining is a technique to implement a hash table
- Number of objects n, hash function cardinality m, longest chain length c
- Memory consumption is $\Theta(n + m)$

Conclusion

- Chaining is a technique to implement a hash table
- Number of objects n, hash function cardinality m, longest chain length c
- Memory consumption is $\Theta(n + m)$
- Operations work in time $\Theta(c + 1)$

Conclusion

- Chaining is a technique to implement a hash table
- Number of objects n, hash function cardinality m, longest chain length c
- Memory consumption is $\Theta(n + m)$
- Operations work in time $\Theta(c + 1)$
- How to make both m and c small?