

# Hashing: Hash Functions

Michael Levin

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

**Data Structures Fundamentals**  
**Algorithms and Data Structures**

# Outline

- 1 Phone Book Data Structure
- 2 Universal Family
- 3 Hashing Phone Numbers
- 4 Hashing Names
- 5 Analysis of Polynomial Hashing

# Phone Book

Design a data structure to store your contacts: names of people along with their phone numbers. The following operations should be fast:

- Add and delete contacts,
- Call person by name,
- Determine who is calling given their phone number.

- We need two Maps:  
(phone number  $\rightarrow$  name) and  
(name  $\rightarrow$  phone number)

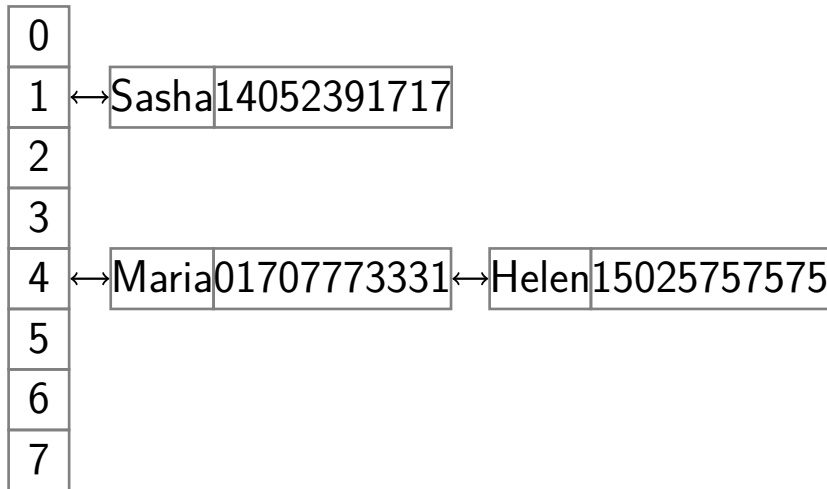
- We need two Maps:  
(phone number  $\rightarrow$  name) and  
(name  $\rightarrow$  phone number)
- Implement these Maps as hash tables

- We need two Maps:  
(phone number  $\rightarrow$  name) and  
(name  $\rightarrow$  phone number)
- Implement these Maps as hash tables
- First, we will focus on the Map from  
phone numbers to names

# Chaining for Phone Book

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

# Chaining for Phone Book





# Parameters

- $n$  contacts stored

# Parameters

- $n$  contacts stored
- $m$  — cardinality of the hash function

# Parameters

- $n$  contacts stored
- $m$  — cardinality of the hash function
- Note: hash table size for chaining should be equal to  $m$

# Parameters

- $n$  contacts stored
- $m$  — cardinality of the hash function
- Note: hash table size for chaining should be equal to  $m$
- $c$  — length of the longest chain

# Parameters

- $n$  contacts stored
- $m$  — cardinality of the hash function
- Note: hash table size for chaining should be equal to  $m$
- $c$  — length of the longest chain
- $\Theta(n + m)$  memory is used

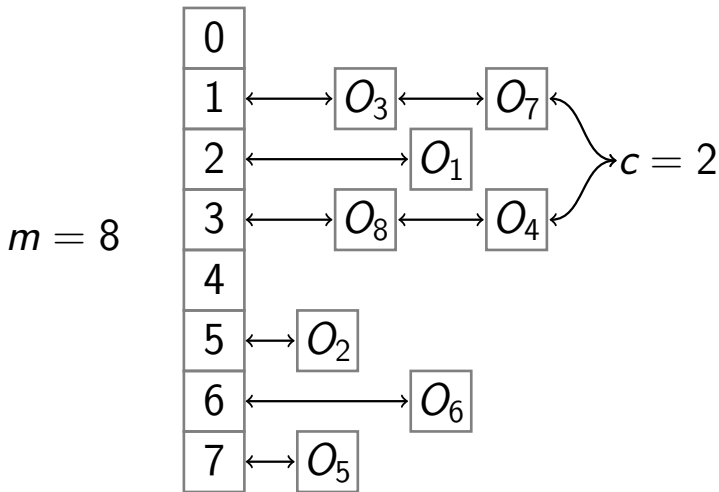
# Parameters

- $n$  contacts stored
- $m$  — cardinality of the hash function
- Note: hash table size for chaining should be equal to  $m$
- $c$  — length of the longest chain
- $\Theta(n + m)$  memory is used
- Operations run in time  $\Theta(c + 1)$

# Parameters

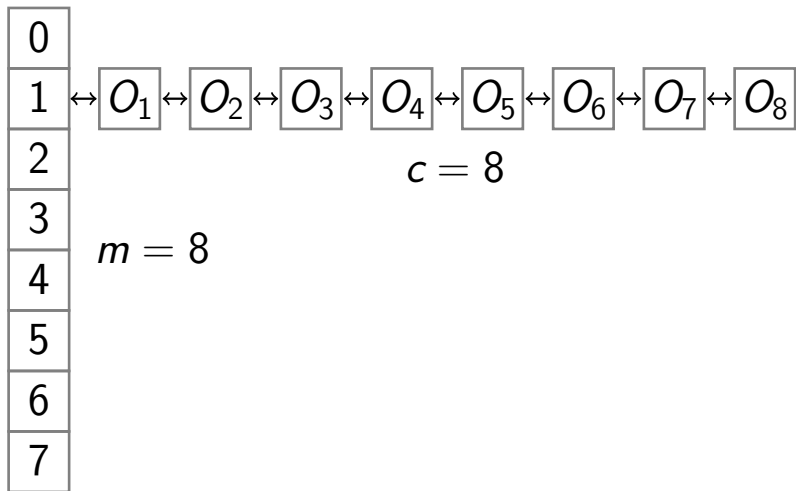
- $n$  contacts stored
- $m$  — cardinality of the hash function
- Note: hash table size for chaining should be equal to  $m$
- $c$  — length of the longest chain
- $\Theta(n + m)$  memory is used
- Operations run in time  $\Theta(c + 1)$
- You want small  $m$  and  $c$ ! (but  $c \geq \frac{n}{m}$ )

# Good Example





# Bad Example



# First Digits

- For the map from phone numbers to names, select  $m = 1000$

# First Digits

- For the map from phone numbers to names, select  $m = 1000$
- Hash function: take first three digits

# First Digits

- For the map from phone numbers to names, select  $m = 1000$
- Hash function: take first three digits
- $h(800-123-45-67) = 800$

# First Digits

- For the map from phone numbers to names, select  $m = 1000$
- Hash function: take first three digits
- $h(800-123-45-67) = 800$
- Problem: area code

# First Digits

- For the map from phone numbers to names, select  $m = 1000$
- Hash function: take first three digits
- $h(800-123-45-67) = 800$
- Problem: area code
- $h(425-234-55-67) =$   
 $h(425-123-45-67) =$   
 $h(425-223-23-23) = \dots = 425$

# Last Digits

- Select  $m = 1000$

# Last Digits

- Select  $m = 1000$
- Hash function: take last three digits



# Last Digits

- Select  $m = 1000$
- Hash function: take last three digits
- $h(800-123-45-67) = 567$

# Last Digits

- Select  $m = 1000$
- Hash function: take last three digits
- $h(800-123-45-67) = 567$
- Problem if many phone numbers end with three zeros

# Random Value

- Select  $m = 1000$

# Random Value

- Select  $m = 1000$
- Hash function: random number between 0 and 999

# Random Value

- Select  $m = 1000$
- Hash function: random number between 0 and 999
- Uniform distribution of hash values

# Random Value

- Select  $m = 1000$
- Hash function: random number between 0 and 999
- Uniform distribution of hash values
- Different value when hash function called again — we won't be able to find anything!

# Random Value

- Select  $m = 1000$
- Hash function: random number between 0 and 999
- Uniform distribution of hash values
- Different value when hash function called again — we won't be able to find anything!
- Hash function must be deterministic

# Good Hash Functions

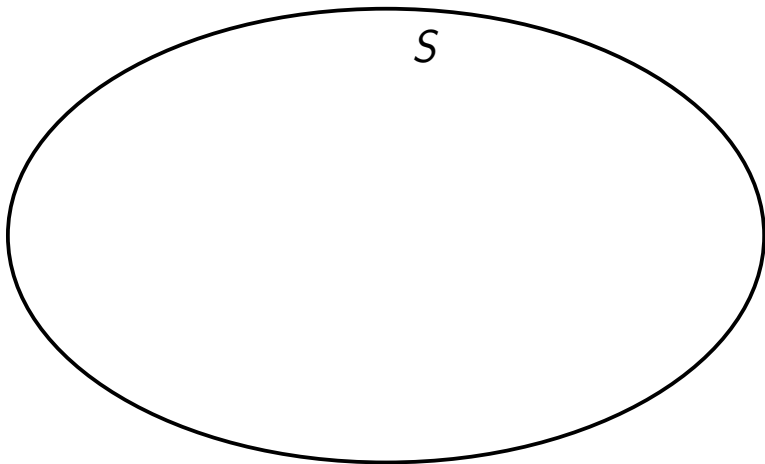
- Deterministic
- Fast to compute
- Distributes keys well into different cells
- Few collisions

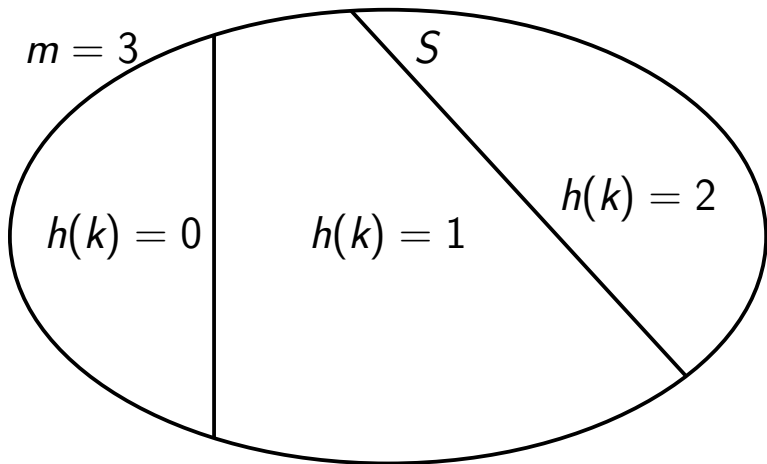


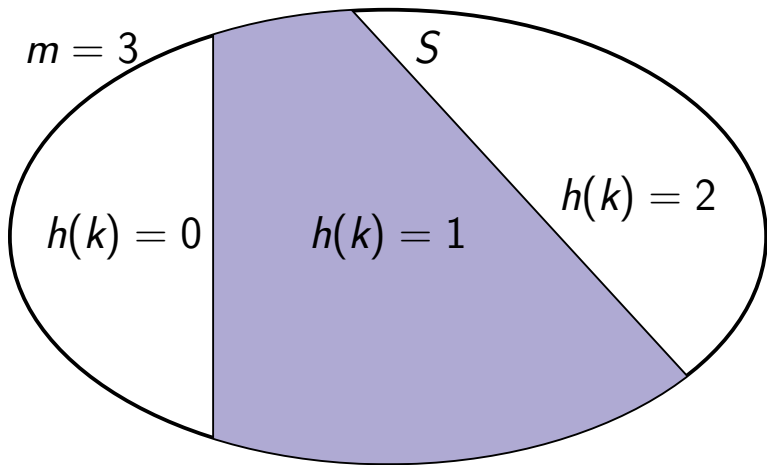
# No Universal Hash Function

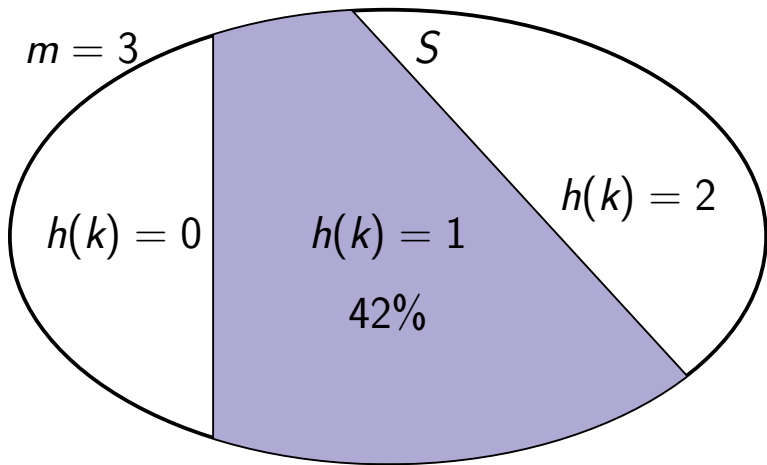
## Lemma

If the number of possible keys is big ( $|S| \gg m$ ), for any hash function  $h$  there is a bad input resulting in many collisions.









# Outline

- 1 Phone Book Data Structure
- 2 Universal Family
- 3 Hashing Phone Numbers
- 4 Hashing Names
- 5 Analysis of Polynomial Hashing

# Idea

- Remember QuickSort?

# Idea

- Remember QuickSort?
- Choosing random pivot helped



# Idea

- Remember QuickSort?
- Choosing random pivot helped
- Use randomization!

# Idea

- Remember QuickSort?
- Choosing random pivot helped
- Use randomization!
- Define a **family** (set) of hash functions

# Idea

- Remember QuickSort?
- Choosing random pivot helped
- Use randomization!
- Define a **family** (set) of hash functions
- Choose random function from the family

# Universal Family

## Definition

Let  $U$  be the **universe** — the set of all possible keys.

# Universal Family

## Definition

Let  $U$  be the **universe** — the set of all possible keys. A set of hash functions

$$\mathcal{H} = \{h : U \rightarrow \{0, 1, 2, \dots, m-1\}\}$$

# Universal Family

## Definition

Let  $U$  be the **universe** — the set of all possible keys. A set of hash functions

$$\mathcal{H} = \{h : U \rightarrow \{0, 1, 2, \dots, m-1\}\}$$

is called a **universal family** if

# Universal Family

## Definition

Let  $U$  be the **universe** — the set of all possible keys. A set of hash functions

$$\mathcal{H} = \{h : U \rightarrow \{0, 1, 2, \dots, m-1\}\}$$

is called a **universal family** if for any two keys  $x, y \in U, x \neq y$  the probability of **collision**

$$\Pr[h(x) = h(y)] \leq \frac{1}{m}$$

# Universal Family

$$\Pr[h(x) = h(y)] \leq \frac{1}{m}$$

means that a collision  $h(x) = h(y)$  for any **fixed** pair of different keys  $x$  and  $y$  happens for no more than  $\frac{1}{m}$  of all hash functions  $h \in \mathcal{H}$ .



# How Randomization Works

- $h(x) = \text{random}(\{0, 1, 2, \dots, m-1\})$   
gives probability of collision exactly  $\frac{1}{m}$ .

# How Randomization Works

- $h(x) = \text{random}(\{0, 1, 2, \dots, m-1\})$   
gives probability of collision exactly  $\frac{1}{m}$ .
- It is not deterministic — can't use it.

# How Randomization Works

- $h(x) = \text{random}(\{0, 1, 2, \dots, m-1\})$   
gives probability of collision exactly  $\frac{1}{m}$ .
- It is not deterministic — can't use it.
- All hash functions in  $\mathcal{H}$  are deterministic

# How Randomization Works

- $h(x) = \text{random}(\{0, 1, 2, \dots, m-1\})$   
gives probability of collision exactly  $\frac{1}{m}$ .
- It is not deterministic — can't use it.
- All hash functions in  $\mathcal{H}$  are deterministic
- Select a random function  $h$  from  $\mathcal{H}$

# How Randomization Works

- $h(x) = \text{random}(\{0, 1, 2, \dots, m-1\})$   
gives probability of collision exactly  $\frac{1}{m}$ .
- It is not deterministic — can't use it.
- All hash functions in  $\mathcal{H}$  are deterministic
- Select a random function  $h$  from  $\mathcal{H}$
- Fixed  $h$  is used throughout the algorithm

# Load Factor

## Definition

The ratio  $\alpha = \frac{n}{m}$  between number of objects  $n$  stored in the hash table and the size of the hash table  $m$  is called **load factor**.

# Running Time

## Lemma

If  $h$  is chosen randomly from a **universal family**, the average length of the longest chain  $c$  is  $O(1 + \alpha)$ , where  $\alpha = \frac{n}{m}$  is the **load factor** of the hash table.

## Corollary

*If  $h$  is from **universal family**, operations with hash table run on average in time  $O(1 + \alpha)$ .*

# Choosing Hash Table Size

- Control amount of memory used with  $m$



# Choosing Hash Table Size

- Control amount of memory used with  $m$
- Ideally, load factor  $0.5 < \alpha < 1$

# Choosing Hash Table Size

- Control amount of memory used with  $m$
- Ideally, **load factor**  $0.5 < \alpha < 1$
- Use  $O(m) = O(\frac{n}{\alpha}) = O(n)$  memory to store  $n$  keys

# Choosing Hash Table Size

- Control amount of memory used with  $m$
- Ideally, **load factor**  $0.5 < \alpha < 1$
- Use  $O(m) = O(\frac{n}{\alpha}) = O(n)$  memory to store  $n$  keys
- Operations run in time  $O(1 + \alpha) = O(1)$  on average

# Dynamic Hash Tables

- What if number of keys  $n$  is unknown in advance?

# Dynamic Hash Tables

- What if number of keys  $n$  is unknown in advance?
- Start with very big hash table?

# Dynamic Hash Tables

- What if number of keys  $n$  is unknown in advance?
- Start with very big hash table?
- You will waste a lot of memory

# Dynamic Hash Tables

- What if number of keys  $n$  is unknown in advance?
- Start with very big hash table?
- You will waste a lot of memory
- Copy the idea of dynamic arrays!

# Dynamic Hash Tables

- What if number of keys  $n$  is unknown in advance?
- Start with very big hash table?
- You will waste a lot of memory
- Copy the idea of dynamic arrays!
- Resize the hash table when  $\alpha$  becomes too large



# Dynamic Hash Tables

- What if number of keys  $n$  is unknown in advance?
- Start with very big hash table?
- You will waste a lot of memory
- Copy the idea of dynamic arrays!
- Resize the hash table when  $\alpha$  becomes too large
- Choose new hash function and **rehash** all the objects

# Rehashing

Keep **load factor** below 0.9:

## Rehash( $T$ )

$loadFactor \leftarrow \frac{T.numberOfKeys}{T.size}$

if  $loadFactor > 0.9$ :

    Create  $T_{new}$  of size  $2 \times T.size$

    Choose  $h_{new}$  with cardinality  $T_{new}.size$

    For each object in  $T$ :

        Insert object in  $T_{new}$  using  $h_{new}$

$T \leftarrow T_{new}, h \leftarrow h_{new}$

# Rehashing

Keep **load factor** below 0.9:

## Rehash( $T$ )

$loadFactor \leftarrow \frac{T.numberOfKeys}{T.size}$

if  $loadFactor > 0.9$ :

    Create  $T_{new}$  of size  $2 \times T.size$

    Choose  $h_{new}$  with cardinality  $T_{new}.size$

    For each object in  $T$ :

        Insert object in  $T_{new}$  using  $h_{new}$

$T \leftarrow T_{new}, h \leftarrow h_{new}$

# Rehashing

Keep **load factor** below 0.9:

## Rehash( $T$ )

$loadFactor \leftarrow \frac{T.numberOfKeys}{T.size}$

**if**  $loadFactor > 0.9$ :

    Create  $T_{new}$  of size  $2 \times T.size$

    Choose  $h_{new}$  with cardinality  $T_{new}.size$

    For each object in  $T$ :

        Insert object in  $T_{new}$  using  $h_{new}$

$T \leftarrow T_{new}, h \leftarrow h_{new}$

# Rehashing

Keep **load factor** below 0.9:

## Rehash( $T$ )

$loadFactor \leftarrow \frac{T.numberOfKeys}{T.size}$

if  $loadFactor > 0.9$ :

    Create  $T_{new}$  of size  $2 \times T.size$

    Choose  $h_{new}$  with cardinality  $T_{new}.size$

    For each object in  $T$ :

        Insert object in  $T_{new}$  using  $h_{new}$

$T \leftarrow T_{new}, h \leftarrow h_{new}$

# Rehashing

Keep **load factor** below 0.9:

## Rehash( $T$ )

$loadFactor \leftarrow \frac{T.numberOfKeys}{T.size}$

if  $loadFactor > 0.9$ :

    Create  $T_{new}$  of size  $2 \times T.size$

    Choose  $h_{new}$  with cardinality  $T_{new}.size$

    For each object in  $T$ :

        Insert object in  $T_{new}$  using  $h_{new}$

$T \leftarrow T_{new}, h \leftarrow h_{new}$

# Rehashing

Keep **load factor** below 0.9:

## Rehash( $T$ )

$loadFactor \leftarrow \frac{T.numberOfKeys}{T.size}$

if  $loadFactor > 0.9$ :

    Create  $T_{new}$  of size  $2 \times T.size$

    Choose  $h_{new}$  with cardinality  $T_{new}.size$

    For each object in  $T$ :

        Insert object in  $T_{new}$  using  $h_{new}$

$T \leftarrow T_{new}, h \leftarrow h_{new}$

# Rehashing

Keep **load factor** below 0.9:

## Rehash( $T$ )

$loadFactor \leftarrow \frac{T.numberOfKeys}{T.size}$

if  $loadFactor > 0.9$ :

    Create  $T_{new}$  of size  $2 \times T.size$

    Choose  $h_{new}$  with cardinality  $T_{new}.size$

    For each object in  $T$ :

        Insert object in  $T_{new}$  using  $h_{new}$

$T \leftarrow T_{new}, h \leftarrow h_{new}$



# Rehashing

Keep **load factor** below 0.9:

## Rehash( $T$ )

$loadFactor \leftarrow \frac{T.numberOfKeys}{T.size}$

if  $loadFactor > 0.9$ :

    Create  $T_{new}$  of size  $2 \times T.size$

    Choose  $h_{new}$  with cardinality  $T_{new}.size$

    For each object in  $T$ :

        Insert object in  $T_{new}$  using  $h_{new}$

$T \leftarrow T_{new}, h \leftarrow h_{new}$

# Rehash Running Time

You should call `Rehash` after each operation with the hash table

Similarly to dynamic arrays, single rehashing takes  $O(n)$  time, but amortized running time of each operation with hash table is still  $O(1)$  on average, because rehashing will be rare

# Outline

- 1 Phone Book Data Structure
- 2 Universal Family
- 3 Hashing Phone Numbers**
- 4 Hashing Names
- 5 Analysis of Polynomial Hashing

# Hashing Phone Numbers

- We can convert phone numbers to integers, for example  
 $148-25-67 \rightarrow 1482567$

# Hashing Phone Numbers

- We can convert phone numbers to integers, for example  
 $148-25-67 \rightarrow 1482567$
- As a result, any phone number will be converted to an integer less than  $10^{15}$

# Hashing Phone Numbers

- We can convert phone numbers to integers, for example  
 $148-25-67 \rightarrow 1482567$
- As a result, any phone number will be converted to an integer less than  $10^{15}$
- If we come up with a **universal family** for integers up to  $10^{15}$ , we will be able to map phone numbers to names efficiently using chaining

# Hashing Integers

## Lemma

$\mathcal{H}_p = \left\{ h_p^{a,b}(x) = ((ax + b) \bmod p) \bmod m \right\}$   
for all  $a, b : 1 \leq a \leq p - 1, 0 \leq b \leq p - 1$  is  
a **universal family** for the set of integers  
between 0 and  $p - 1$ , for any prime  $p$ .

# Hashing Phone Numbers

## Example

Select  $a = 34$ ,  $b = 2$ , so  $h = h_p^{34,2}$  and consider  $x = 1\ 482\ 567$  corresponding to phone number 148-25-67.  $p = 10\ 000\ 019$ .



# Hashing Phone Numbers

## Example

Select  $a = 34$ ,  $b = 2$ , so  $h = h_p^{34,2}$  and consider  $x = 1\ 482\ 567$  corresponding to phone number 148-25-67.  $p = 10\ 000\ 019$ .

$$(34 \times 1482567 + 2) \bmod 10000019 = 407185$$

# Hashing Phone Numbers

## Example

Select  $a = 34$ ,  $b = 2$ , so  $h = h_p^{34,2}$  and consider  $x = 1\ 482\ 567$  corresponding to phone number 148-25-67.  $p = 10\ 000\ 019$ .

$$(34 \times 1482567 + 2) \bmod 10000019 = 407185$$

$$407185 \bmod 1000 = 185$$

# Hashing Phone Numbers

## Example

Select  $a = 34$ ,  $b = 2$ , so  $h = h_p^{34,2}$  and consider  $x = 1\ 482\ 567$  corresponding to phone number 148-25-67.  $p = 10\ 000\ 019$ .

$$(34 \times 1482567 + 2) \bmod 10000019 = 407185$$

$$407185 \bmod 1000 = 185$$

$$h(x) = 185$$

# Proof Ideas

- For any pair of different keys  $(x, y)$ , any two of the  $p(p - 1)$  hash functions in  $\mathcal{H}_p$  hash them into different pairs  $(r, s)$  of different remainders modulo  $p$

# Proof Ideas

- For any pair of different keys  $(x, y)$ , any two of the  $p(p - 1)$  hash functions in  $\mathcal{H}_p$  hash them into different pairs  $(r, s)$  of different remainders modulo  $p$
- Thus any pair  $(r, s)$  of different remainders modulo  $p$  has equal probability  $\frac{1}{p(p-1)}$

# Proof Ideas

- For any pair of different keys  $(x, y)$ , any two of the  $p(p - 1)$  hash functions in  $\mathcal{H}_p$  hash them into different pairs  $(r, s)$  of different remainders modulo  $p$
- Thus any pair  $(r, s)$  of different remainders modulo  $p$  has equal probability  $\frac{1}{p(p-1)}$
- The ratio of pairs  $(r, s)$  of different remainders modulo  $p$  such that  $r \equiv s \pmod{m}$  is less than  $\frac{1}{m}$



# General Case

- Define maximum length  $L$  of a phone number

# General Case

- Define maximum length  $L$  of a phone number
- Convert phone numbers to integers from 0 to  $10^L - 1$



# General Case

- Define maximum length  $L$  of a phone number
- Convert phone numbers to integers from 0 to  $10^L - 1$
- Choose prime number  $p > 10^L$

# General Case

- Define maximum length  $L$  of a phone number
- Convert phone numbers to integers from 0 to  $10^L - 1$
- Choose prime number  $p > 10^L$
- Choose hash table size  $m$

# General Case

- Define maximum length  $L$  of a phone number
- Convert phone numbers to integers from 0 to  $10^L - 1$
- Choose prime number  $p > 10^L$
- Choose hash table size  $m$
- Choose random hash function from **universal family**  $\mathcal{H}_p$  (choose random  $a \in [1, p - 1]$  and  $b \in [0, p - 1]$ )

# Outline

- 1 Phone Book Data Structure
- 2 Universal Family
- 3 Hashing Phone Numbers
- 4 Hashing Names**
- 5 Analysis of Polynomial Hashing

# Lookup Phone Numbers by Name

- Time to implement Map from names to phone numbers

# Lookup Phone Numbers by Name

- Time to implement Map from names to phone numbers
- Use chaining again

# Lookup Phone Numbers by Name

- Time to implement Map from names to phone numbers
- Use chaining again
- Need a hash function for names

# Lookup Phone Numbers by Name

- Time to implement Map from names to phone numbers
- Use chaining again
- Need a hash function for names
- Hash arbitrary strings of bounded length



# Lookup Phone Numbers by Name

- Time to implement Map from names to phone numbers
- Use chaining again
- Need a hash function for names
- Hash arbitrary strings of bounded length
- You will learn how string hashing is implemented in Java!

# String Length Notation

## Definition

Denote by  $|S|$  the length of string  $S$ .

## Examples

$$|\text{"edx"}| = 3$$

$$|\text{"ucsd"}| = 4$$

$$|\text{"chaining"}| = 8$$

# Hashing Strings

- Given a string  $S$ , compute its hash value

# Hashing Strings

- Given a string  $S$ , compute its hash value
- $S = S[0]S[1] \dots S[|S| - 1]$ , where  $S[i]$  are individual characters

# Hashing Strings

- Given a string  $S$ , compute its hash value
- $S = S[0]S[1] \dots S[|S| - 1]$ , where  $S[i]$  are individual characters
- We should use all the characters in the hash function

# Hashing Strings

- Given a string  $S$ , compute its hash value
- $S = S[0]S[1] \dots S[|S| - 1]$ , where  $S[i]$  are individual characters
- We should use all the characters in the hash function
- Otherwise there will be many collisions

# Hashing Strings

- Given a string  $S$ , compute its hash value
- $S = S[0]S[1] \dots S[|S| - 1]$ , where  $S[i]$  are individual characters
- We should use all the characters in the hash function
- Otherwise there will be many collisions
- For example, if  $S[0]$  is not used, then  $h(\text{"aa"}) = h(\text{"ba"}) = \dots = h(\text{"za"})$

# Preparation

- Convert each character  $S[i]$  to integer



# Preparation

- Convert each character  $S[i]$  to integer
- ASCII encoding, Unicode, etc.

# Preparation

- Convert each character  $S[i]$  to integer
- ASCII encoding, Unicode, etc.
- Choose big prime number  $p$

# Polynomial Hashing

## Definition

Family of hash functions

$$\mathcal{P}_p = \left\{ h_p^x(S) = \sum_{i=0}^{|S|-1} S[i]x^i \bmod p \right\}$$

with a fixed prime  $p$  and all  $1 \leq x \leq p-1$  is called **polynomial**.

## PolyHash( $S, p, x$ )

hash  $\leftarrow 0$

for  $i$  from  $|S| - 1$  down to 0:

    hash  $\leftarrow (\text{hash} \cdot x + S[i]) \bmod p$

return hash

## PolyHash( $S, p, x$ )

hash  $\leftarrow 0$

for  $i$  from  $|S| - 1$  down to 0:

    hash  $\leftarrow (\text{hash} \cdot x + S[i]) \bmod p$

return hash

## PolyHash( $S, p, x$ )

hash  $\leftarrow 0$

for  $i$  from  $|S| - 1$  down to 0:

    hash  $\leftarrow (\text{hash} \cdot x + S[i]) \bmod p$

return hash

## PolyHash( $S, p, x$ )

hash  $\leftarrow 0$

for  $i$  from  $|S| - 1$  down to 0:

    hash  $\leftarrow (\text{hash} \cdot x + S[i]) \bmod p$

return hash

## PolyHash( $S, p, x$ )

hash  $\leftarrow 0$

for  $i$  from  $|S| - 1$  down to 0:

    hash  $\leftarrow (\text{hash} \cdot x + S[i]) \bmod p$

return hash



## PolyHash( $S, p, x$ )

hash  $\leftarrow 0$

for  $i$  from  $|S| - 1$  down to 0:

    hash  $\leftarrow (\text{hash} \cdot x + S[i]) \bmod p$

return hash

Example:  $|S| = 3$

## PolyHash( $S, p, x$ )

hash  $\leftarrow 0$

for  $i$  from  $|S| - 1$  down to 0:

    hash  $\leftarrow (\text{hash} \cdot x + S[i]) \bmod p$

return hash

Example:  $|S| = 3$

1 hash  $\leftarrow 0$

## PolyHash( $S, p, x$ )

hash  $\leftarrow 0$

for  $i$  from  $|S| - 1$  down to 0:

    hash  $\leftarrow (\text{hash} \cdot x + S[i]) \bmod p$

return hash

Example:  $|S| = 3$

1 hash  $\leftarrow 0$

2 hash  $\leftarrow S[2] \bmod p$

## PolyHash( $S, p, x$ )

hash  $\leftarrow 0$

for  $i$  from  $|S| - 1$  down to 0:

    hash  $\leftarrow (\text{hash} \cdot x + S[i]) \bmod p$

return hash

Example:  $|S| = 3$

1 hash  $\leftarrow 0$

2 hash  $\leftarrow S[2] \bmod p$

3 hash  $\leftarrow S[1] + S[2]x \bmod p$

## PolyHash( $S, p, x$ )

hash  $\leftarrow 0$

for  $i$  from  $|S| - 1$  down to 0:

    hash  $\leftarrow (\text{hash} \cdot x + S[i]) \bmod p$

return hash

Example:  $|S| = 3$

1 hash  $\leftarrow 0$

2 hash  $\leftarrow S[2] \bmod p$

3 hash  $\leftarrow S[1] + S[2]x \bmod p$

4 hash  $\leftarrow S[0] + S[1]x + S[2]x^2 \bmod p$

# Java Implementation

The method `hashCode` of the built-in Java class `String` is very similar to our `PolyHash`, it just uses  $x = 31$  and for technical reasons avoids the  $(\bmod p)$  operator.

# Java Implementation

The method `hashCode` of the built-in Java class `String` is very similar to our `PolyHash`, it just uses  $x = 31$  and for technical reasons avoids the  $(\text{mod } p)$  operator.

You now know the implementation of the function that is used trillions of times a day in many thousands of programs!

# Outline

- 1 Phone Book Data Structure
- 2 Universal Family
- 3 Hashing Phone Numbers
- 4 Hashing Names
- 5 Analysis of Polynomial Hashing



## Lemma

For any two different strings  $s_1$  and  $s_2$  of length at most  $L + 1$ , if you choose  $h$  from  $\mathcal{P}_p$  at random (by selecting a random  $x \in [1, p - 1]$ ), the probability of collision  $\Pr[h(s_1) = h(s_2)]$  is at most  $\frac{L}{p}$ .

## Lemma

For any two different strings  $s_1$  and  $s_2$  of length at most  $L + 1$ , if you choose  $h$  from  $\mathcal{P}_p$  at random (by selecting a random  $x \in [1, p - 1]$ ), the probability of collision  $\Pr[h(s_1) = h(s_2)]$  is at most  $\frac{L}{p}$ .

## Proof idea

This follows from the fact that the equation  $a_0 + a_1x + a_2x^2 + \cdots + a_Lx^L = 0 \pmod{p}$  for prime  $p$  has at most  $L$  different solutions  $x$ .

# Cardinality Fix

For use in a hash table of size  $m$ , we need a hash function of cardinality  $m$ .

# Cardinality Fix

For use in a hash table of size  $m$ , we need a hash function of cardinality  $m$ .

First, apply random  $h_x$  from  $\mathcal{P}_p$ , and then hash the resulting value again using universal family for integers. Denote the resulting function by  $h_m$ .

# Cardinality Fix

For use in a hash table of size  $m$ , we need a hash function of cardinality  $m$ .

First, apply random  $h_x$  from  $\mathcal{P}_p$ , and then hash the resulting value again using universal family for integers. Denote the resulting function by  $h_m$ .

$$h_m(S) = h_{a,b}(h_x(S)) \bmod m$$

# Cardinality Fix

For use in a hash table of size  $m$ , we need a hash function of cardinality  $m$ .

First, apply random  $h_x$  from  $\mathcal{P}_p$ , and then hash the resulting value again using universal family for integers. Denote the resulting function by  $h_m$ .

$$h_m(S) = h_{a,b}(h_x(S)) \bmod m$$

# Cardinality Fix

For use in a hash table of size  $m$ , we need a hash function of cardinality  $m$ .

First, apply random  $h_x$  from  $\mathcal{P}_p$ , and then hash the resulting value again using universal family for integers. Denote the resulting function by  $h_m$ .

$$h_m(S) = h_{a,b}(h_x(S)) \bmod m$$

# Cardinality Fix

For use in a hash table of size  $m$ , we need a hash function of cardinality  $m$ .

First, apply random  $h_x$  from  $\mathcal{P}_p$ , and then hash the resulting value again using universal family for integers. Denote the resulting function by  $h_m$ .

$$h_m(S) = h_{a,b}(h_x(S)) \bmod m$$



## Lemma

For any two different strings  $s_1$  and  $s_2$  of length at most  $L + 1$  and cardinality  $m$ , the probability of collision  $\Pr[h_m(s_1) = h_m(s_2)]$  is at most  $\frac{1}{m} + \frac{L}{p}$ .

# Polynomial Hashing

## Corollary

*If  $p > mL$ , for any two different strings  $s_1$  and  $s_2$  of length at most  $L + 1$  the probability of collision  $\Pr[h_m(s_1) = h_m(s_2)]$  is  $O(\frac{1}{m})$ .*

## Proof

$$\frac{1}{m} + \frac{L}{p} < \frac{1}{m} + \frac{L}{mL} = \frac{1}{m} + \frac{1}{m} = \frac{2}{m} = O\left(\frac{1}{m}\right) \quad \square$$

# Running Time

- For  $p > mL$  we have

$$c = O(1 + \frac{n}{m}) = O(1 + \alpha) \text{ again}$$

# Running Time

- For  $p > mL$  we have  
 $c = O(1 + \frac{n}{m}) = O(1 + \alpha)$  again
- Computing  $\text{PolyHash}(S)$  runs in  $O(|S|)$

# Running Time

- For  $p > mL$  we have  
 $c = O(1 + \frac{n}{m}) = O(1 + \alpha)$  again
- Computing  $\text{PolyHash}(S)$  runs in  $O(|S|)$
- If lengths of the names in the phone book are bounded by constant  $L$ ,  
computing  $h(S)$  takes  $O(L) = O(1)$   
time

# Conclusion

- You learned how to hash integers and strings
- Phone book can be implemented as two hash tables
- Mapping phone numbers to names and back
- Search and modification run in  $O(1)$  on average!