Chapter 13:
The Cloakroom, the Pigeon, and the Bucket

Panos Louridas

Athens University of Economics and Business
Real World Algorithms
A Beginners Guide
The MIT Press

Overview

Outline

Cloakroom

- When you hand in your coat or bag to a cloakroom attendant you get a ticket in return.
- When you want to retrieve your belongings, you hand over your ticket and your item is handed over to you in return.
- If we think about it, this is technique of *locating without searching*.

Locating without Searching

- When we want to store an item, we just calculate the address where we will store it.
- When we want to retrieve an item, again we calculate its address, so that we can fetch it directly from there.

Implementing Locating without Searching (1)

- We want to be able to store and retrieve records.
- Record storage and retrieval will be performed based on their keys.
- We want, having the key of a record, to calculate the address where it will be stored.

Implementing Locating without Searching (2)

- The address of the record will be a memory address in the computer.
- Therefore, it will be just a number.
- We need a function that will convert keys to such addresses.

Implementing Locating without Searching (3)

- We need a function, say $f(K)$, which takes as input the key K of a record R and returns a value a.
- The value a will be the address where R will be stored.
- Every time we want to retrieve R based on its key, we will recalculate $f(K)$, which will return the same address a, where we will find R.

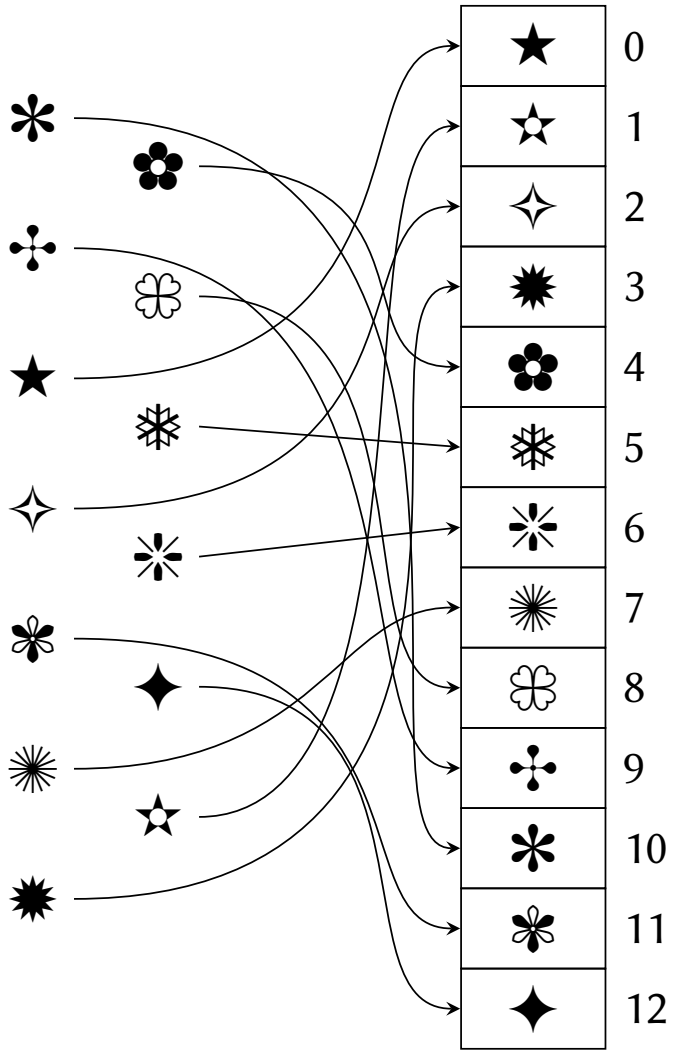
Essential Requirement

- In order for this scheme to work, the function $f(K)$ must be fast.
- If it is fast, then retrieval can be based on the function.
- If it is slow, then it is not worth it.

More on $f(K)$

- Suppose that we have n different records, known in advance.
- Then the problem is finding a function $f(K)$ that will return a different value from 0 up to $n-1$ for each different key.
- With that, we can store each record R in an array T of size n, so that if $f(K) = a$, where K is the key of R, then $T[a] = R$.

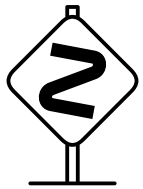
Example



Creating a Function $f(K)$

- Creating such a function $f(K)$ for a set of records is not an easy task.
- We can craft a function $f(K)$ "by hand," but it is difficult and tiresome.
- In particular we can create, if we persist, a function that maps each record to a different address.
- Such a function is a *perfect mapping*, as it maps each input to a different output value.

Attention



The handmade function that follows is just an example; you are not required to learn it.

Perfect Matching Example

Algorithm: A perfect mapping of 31 of most common words in English to numbers.

`PerfectMapping`$(s) \rightarrow r$

Input: s, a string among a predefined list of 31 of the most common words in English

Output: r, an address in the range $-10, 1, \ldots, 29$

```
1  r ← −Code(s[0])
2  s ← Code(s[1])
3  r ← r − 8 + s
4  if r ≤ 0 then
5      r ← r + 16 + s
6  s ←Code(s[2])
7  if s = 0 then
8      return r
9  r ← r − 28 + s
10 if r > 0 then
11     return r
12 r ← r + 11 + s
13 t ← Code(s[3])
14 if t = 0 then
15     return r
16 r ← r − (s − 5)
17 if r < 0 then
18     return r
19 r ← r + 10
20 return r
```

Explanation

- The Code function takes a character and returns a numerical value. The space character is zero, “A” is one, “B” two, etc.
- We add one to that value if it is greater than nine and another two if the resulting value is greater than 19.
- We mangle together the first three characters of the word (padded with spaces, if necessary), and a unique value is produced for the word.

Perfect Mapping of the 31 Most Common English Words

A	7	FOR	23	IN	29	THE	-6
AND	-3	FROM	19	IS	5	THIS	-2
ARE	3	HAD	-7	IT	6	TO	17
AS	13	HAVE	25	NOT	20	WAS	11
AT	14	HE	10	OF	4	WHICH	-5
BE	16	HER	1	ON	22	WITH	21
BUT	9	HIS	12	OR	30	YOU	8
BY	18	I	-1	THAT	-10		

Mapping Problems

- `PerfectMapping` works and is very fast. We only need to add 10 to the result to get a position in an array of $0, 1, \ldots, 39$.
- But it is very obscure.
- We cannot be searching for such functions for each different set of values that we want to store.
- Moreover, if just one key changes, it is possible that the function will no longer work, if it maps two keys to the same value.
- Finally, in the range $-10, -9, \ldots, 29$ there are nine positions that we do not use—so we are wasting some space.

Towards a Different Mapping Function

- Another idea is to take the first character of the key, the last character of the key, and the number of characters of the key.
- Then we can calculate a numerical value as follows:

$$h = \mathsf{Code}(b) + \mathsf{Code}(e) + |K|$$

  where K is the key, $|K|$ is the key's length, b is the character at the beginning of the key, and e is the character at the end of the key.
- The value h will be the address in the array where we will store the record.
- The problem them is finding the right Code function.

Minimal Perfect Mapping

Algorithm: A minimal perfect mapping.

`MinimalPerfectMapping`$(s) \longrightarrow r$

Input: s, a string among a predefined list of 31 of the most common words in English

Output: r, an address in the range $1, \ldots, 32$

Data: C, an array of 26 integers

```
1  C ← [
2      3,  23,  -1,  17,   7,  11,  -1,   5,   0,
3     -1,  -1,  -1,  16,  17,   9,  -1,  -1,  13,
4      4,   0,  23,  -1,   8,  -1,   4,  -1
5  ]
6  l ← |s|
7  b ←Ordinal(s[0])
8  e ←Ordinal(s[l − 1])
9  r ← l + C[b] + C[e]
10 return r
```

Explanation

- `Code` is in fact array C.
- Each position of C corresponds to the number that will be assigned to the corresponding letter of the alphabet.
- Position 0 contains the code for "A", position 1 contains the code for "B", etc.
- `Ordinal(`ch`)` returns the position of ch in the alphabet ("A" is 0, "B" is 1, etc.).
- C has several entries equal to -1. These correspond to alphabet letters that do not appear as first or last in our data set (e.g., the character "C").

Minimal Perfect Mapping Example

A	7	FOR	27	IN	19	THE	10
AND	23	FROM	31	IS	6	THIS	8
ARE	13	HAD	25	IT	2	TO	11
AS	9	HAVE	16	NOT	20	WAS	15
AT	5	HE	14	OF	22	WHICH	18
BE	32	HER	21	ON	28	WITH	17
BUT	26	HIS	12	OR	24	YOU	30
BY	29	I	1	THAT	4		

Features of the Algorithm

- It is a better solution than the previous one. The algorithm is simpler and we do not waste space.
- This is a *minimal perfect mapping*, as it maps all words to distinct values, and it maps n words to n distinct values, the minimum possible number.
- But is it a general solution to our problem?
- No—it requires knowing all keys in advance. It may not work for all keys (e.g., keys "ERA" and "ARE" would map to the same value).
- Creating C is a difficult task, requiring specialized algorithms.

Outline

The Problem

- We want a general purpose function that can map keys to values in a given range.
- The function must do that without requiring prior knowledge of the keys.
- The function must avoid “as much as possible”mapping different keys to the same value.
- We say “as much as possible” because if we have an array of size n and $2n$ possible keys, then at least n keys will be mapped on the same values with other keys.

Hashing

- This technique is called *hashing*. It is also called *scatter storage*.
- The function we use for the mapping is called *hash function*.
- The array to which we map the keys is called *hash table*.
- The array cells are called *buckets* or *slots*.

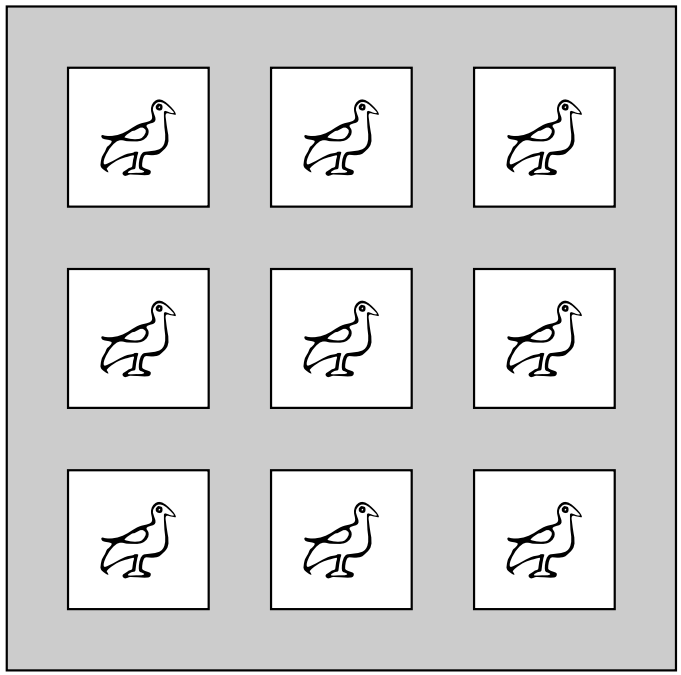
Kinds of Hashing Functions

- If the function maps all keys to different values, it is a *perfect hash function*.
- If the function maps all keys to different values and the range of the generated addresses is equal to the number of the keys, it is a *minimal perfect hash function*.
- If two keys map to the same address, then we have a *collision*.
- If the size of the hash table is smaller than the number of possible keys, then collisions are unavoidable; we want to avoid them as much as possible.

The Pigeonhole Principle

- Collisions are a result of the *pigeonhole principle*.
- If we have n items and m possible containers, with $n > m$, and we want to put all items into the available containers, at least one container will contain more than one item.

The Pigeonhole Principle (Contd.)



?

Hairs

- In any big city there will be at least two people with exactly the same number of hairs on their heads.
- The average number of hairs on a human head is about 150,000.
- The actual number of hairs on a person may of course vary, but we can be sure that there can be no person with, say, 1,000,000 hairs on the head.
- Therefore, in any city with more than 1,000,000 inhabitants there will be at least two people with exactly the same number of hairs on their head.

The Birthday Paradox (1)

- What is the minimum amount of people you need to get into a room so it is probable that two of them have the same birthday?
- Suppose that the probability is $P(B)$.
- We want to find the minimum number of people so that $P(B) > 0.5$.

The Birthday Paradox (2)

- It is easier to calculate the probability $P(\overline{B})$ that there will not be any two people with the same birthday.
- From the laws of probability we know that $P(\overline{B}) = 1 - P(B)$.

The Birthday Paradox (3)

- Let's take each person in turn.
- The first person can have birthday on any day of the year.
- This will happen with probability 365/365.

The Birthday Paradox (4)

- The birthday of the second person must not fall on the same day as the first person's.
- The probability for this is $364/365$.
- So the combined probability for the two persons is $365/365 \times 364/365$.

The Birthday Paradox (5)

- If we go on like this for n persons we will have: $365/365 \times 364/365 \times \cdots \times (365 - n + 1)/365$.
- If we carry out the calculations we will find that for $n = 23$ we have $P(\overline{B}) = 365/365 \times 364/365 \times \cdots \times 343/365 \approx 0.49$.
- So $P(B) \approx 0.51$.
- Therefore if we have 23 people in a room it is more likely that two people will have the same birthday than no two of them will.
- Of course, the more people in the room, the more likely it gets.

Outline

Searching for Hashing Functions

- Suppose that our keys are addresses (street and number) in English.
- Also suppose that the keys are 25 characters long.
- Then the number of possible keys are $25^{37} = 1.6 \times 10^{39}$: each character can be a letter, a digit, or space.
- Of course, the actual keys we will use will be much fewer. Suppose we expect to use 100,000 keys.
- We need a hashing function that can map a range of 1.6×10^{39} to 100,000 addresses of the hash table.
- We will have collisions, but we want to minimize them.

Integer Hashing Function

- If our keys are integers, a hashing function that works well in practice is:

$$h(K) = K \bmod m$$

  where K is the key and m the size of the hash table.
- The function also works for negative keys.
- For any K we have $K \bmod m = r$ where $K = qm + r$ with q equal to $\lfloor K/m \rfloor$. Therefore $r = K - m\lfloor K/m \rfloor$.
- For example, $-6 \bmod 10 = 4$, because $\lfloor -6/10 \rfloor = -1$ and $r = -6 - 10(-1) = -6 + 10 = 4$.

Integer Hashing Algorithm

Algorithm: An integer hash function.

`IntegerHash`$(k, m) \rightarrow h$

Input: k, an integer number

m, the size of the hash table

Output: h, the hash value of k

1 $h \leftarrow k \bmod m$

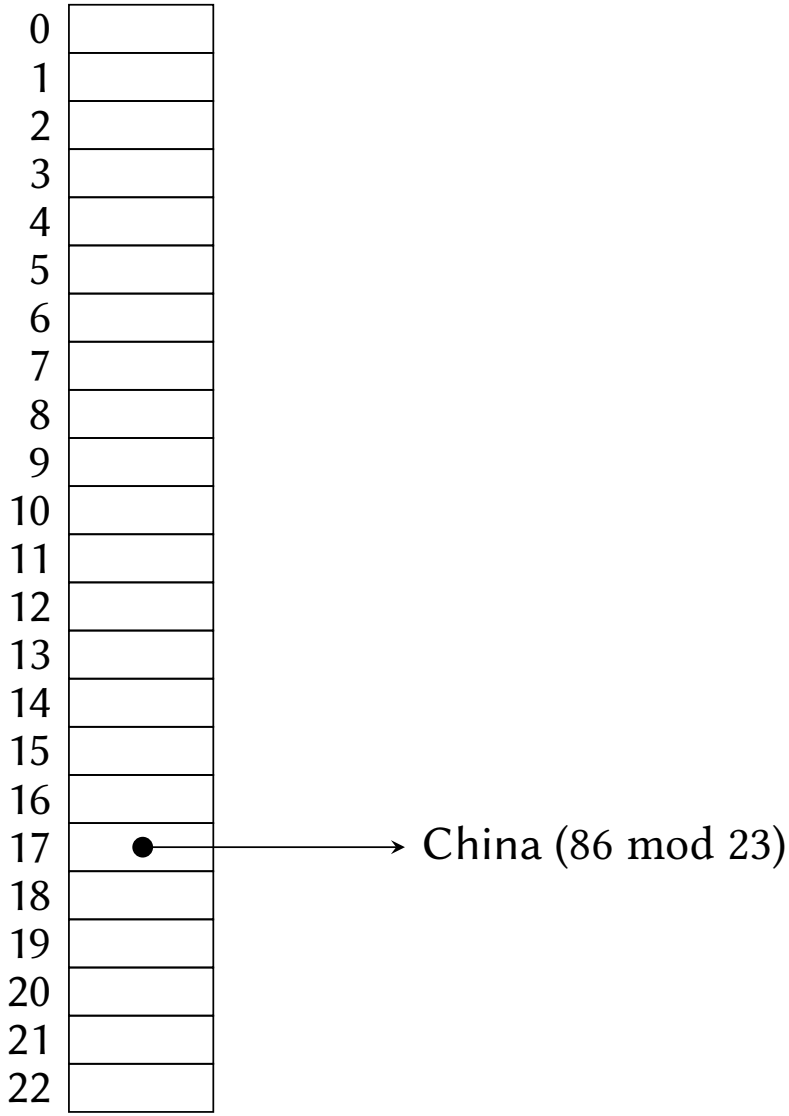
2 **return** h

Integer Hashing Example Data

China	86	Japan	81
India	91	Mexico	52
United States	1	Philippines	63
Indonesia	62	Vietnam	84
Brazil	55	Ethiopia	251
Pakistan	92	Egypt	20
Nigeria	234	Germany	49
Bangladesh	880	Iran	98
Russia	7		

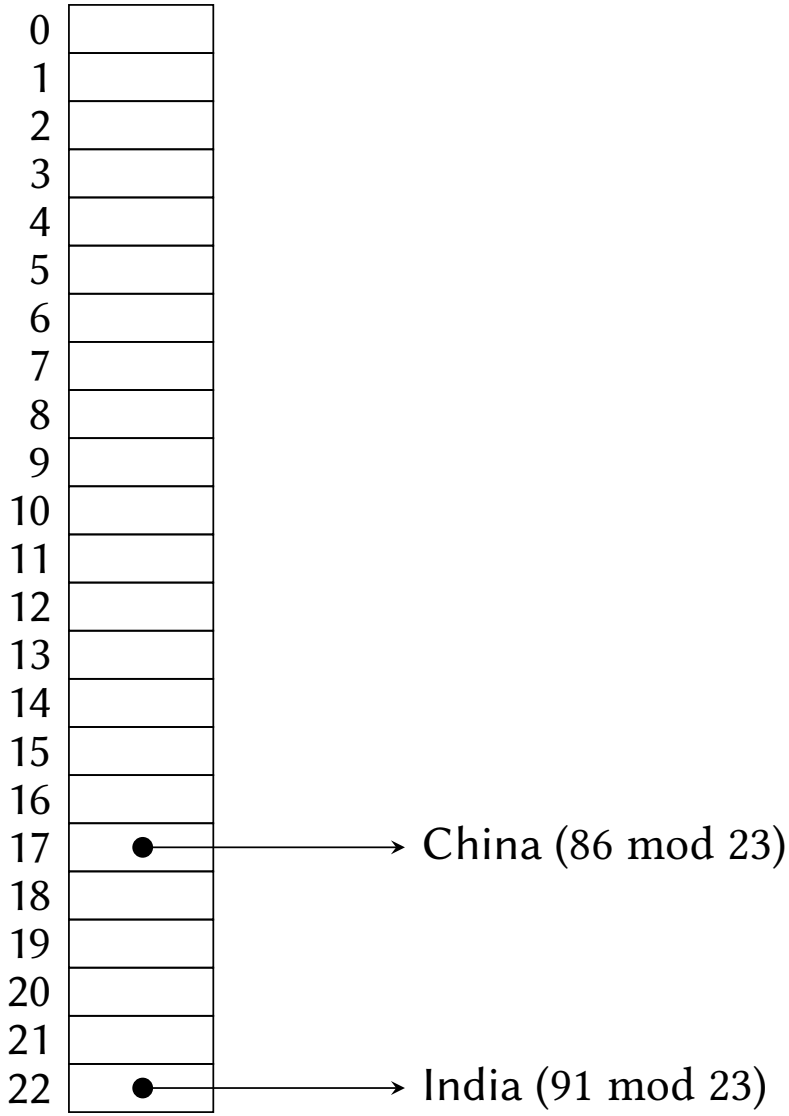
First 17 countries in terms of population with international call codes (2015).

Integer Hashing Example (1)



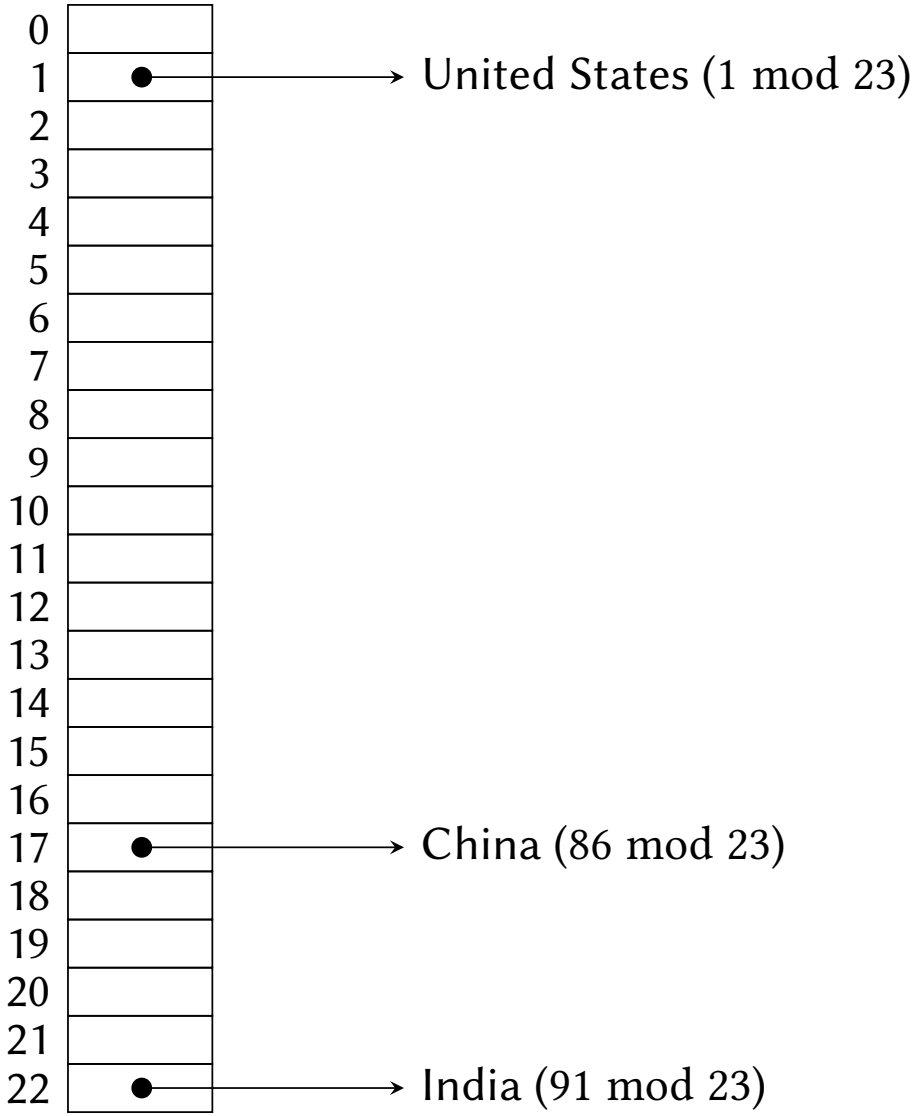
Inserting China.

Integer Hashing Example (2)



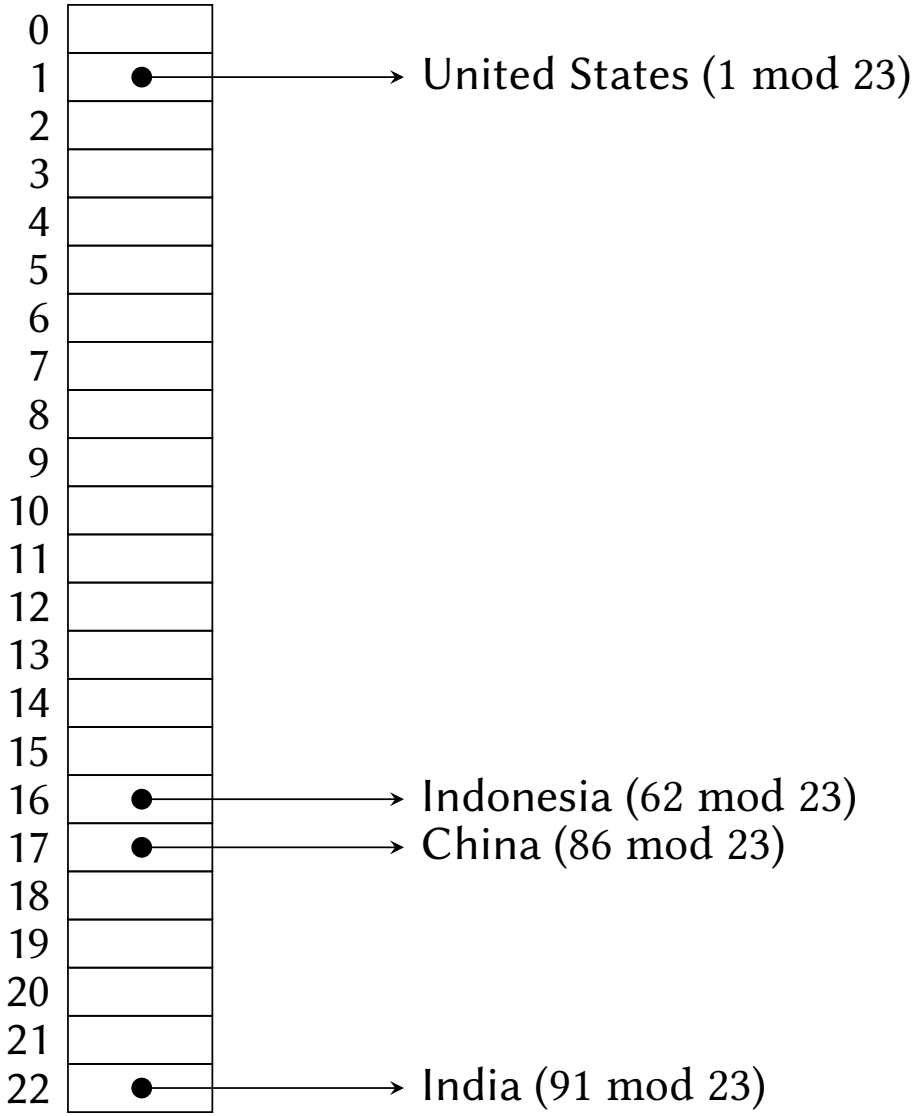
Inserting India.

Integer Hashing (3)



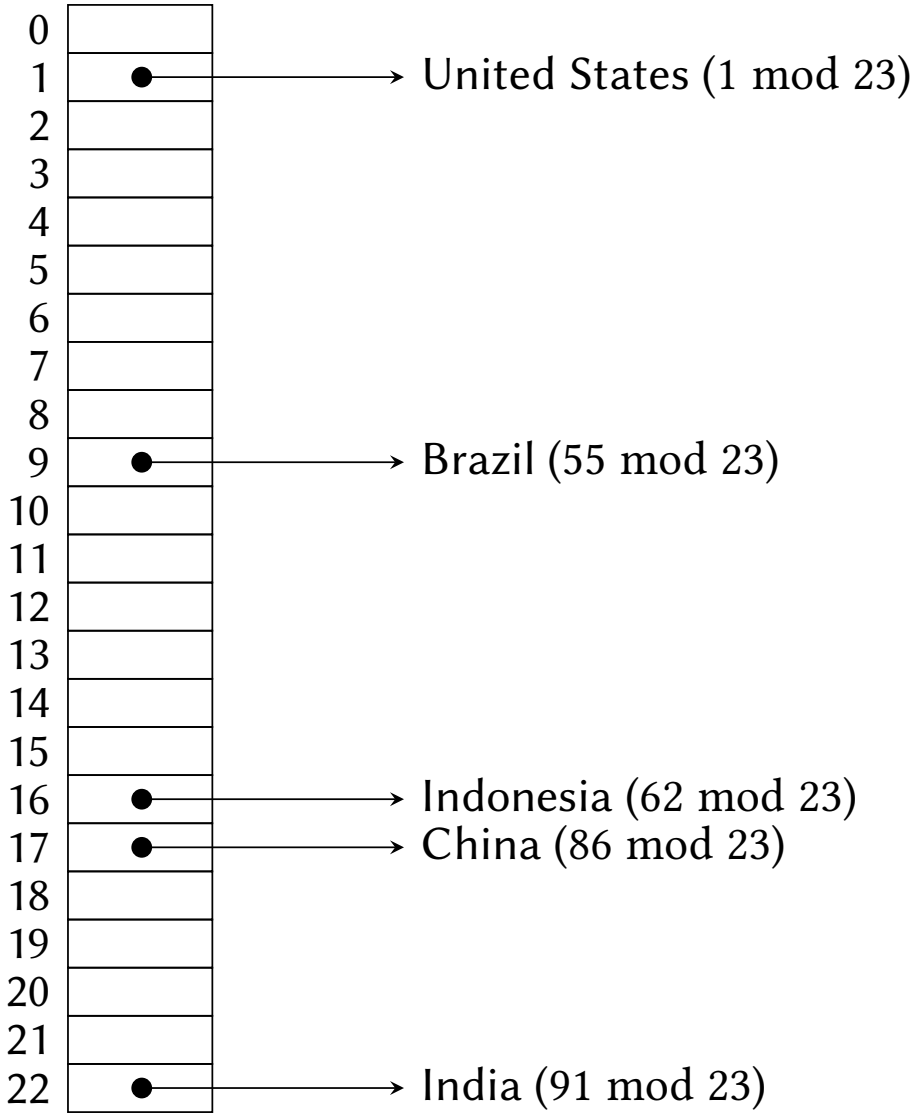
Inserting United States.

Integer Hashing Example (4)



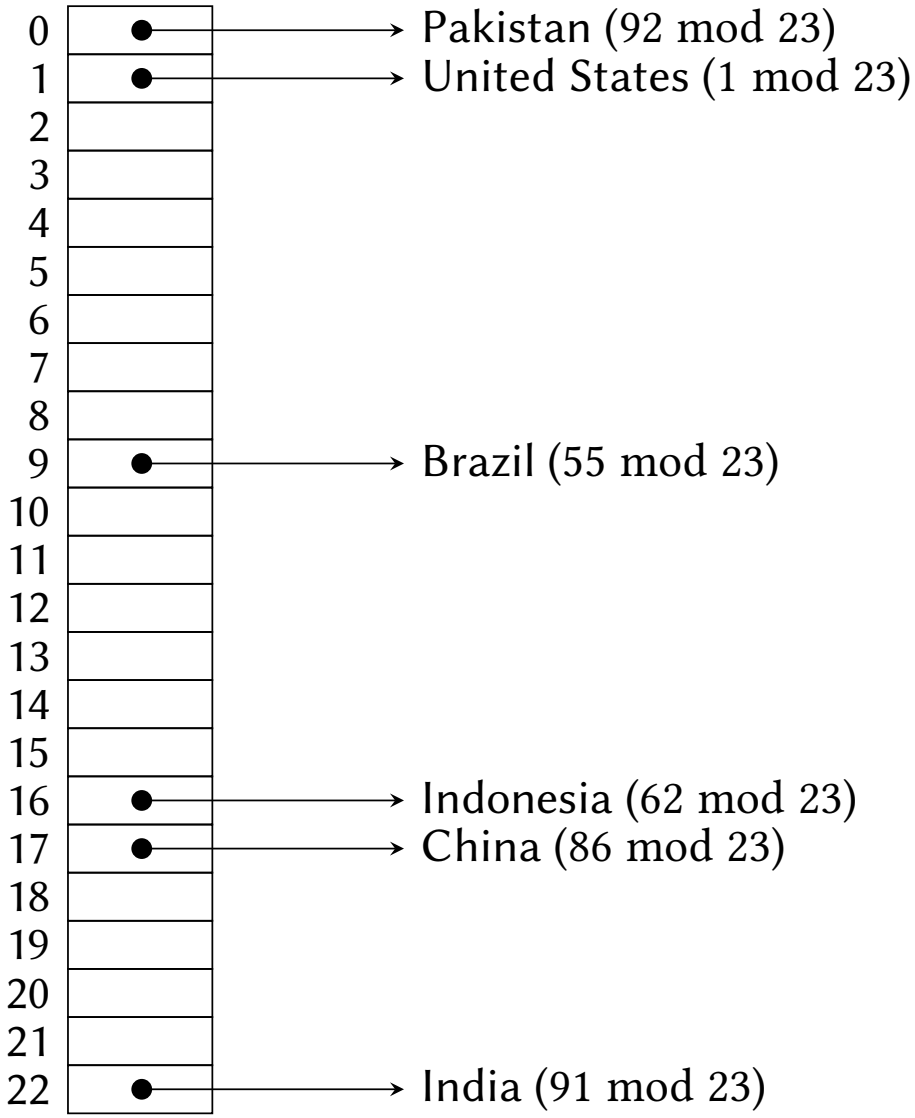
Inserting Indonesia.

Integer Hashing Example (5)



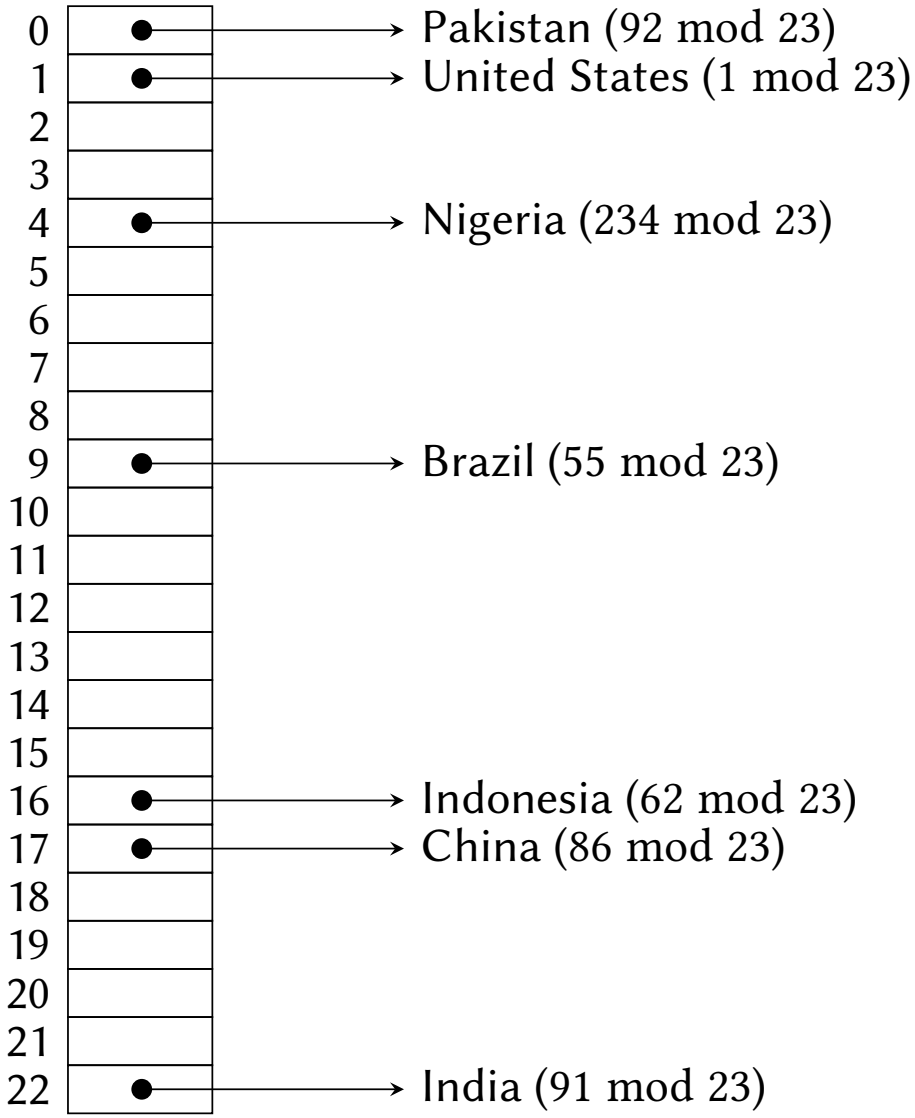
Inserting Brazil.

Integer Hashing Example (6)



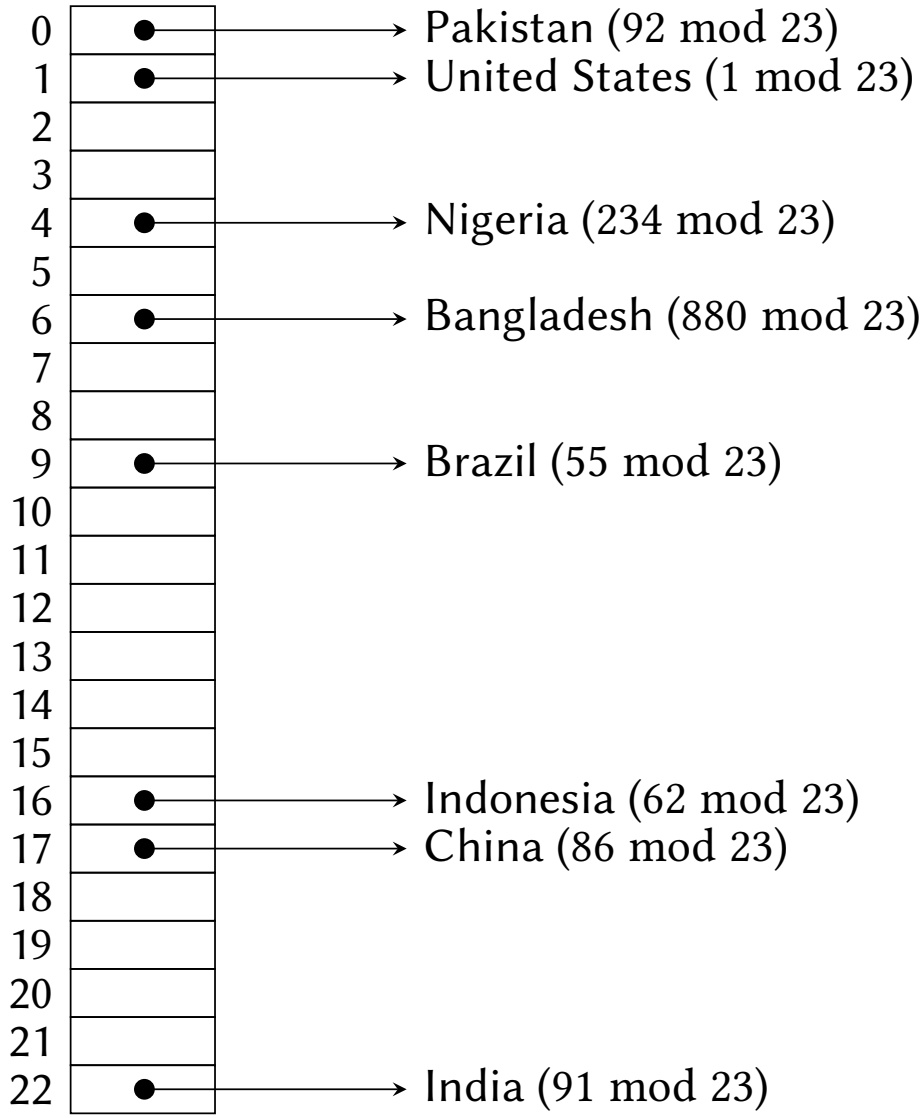
Inserting Pakistan.

Integer Hashing Example (7)



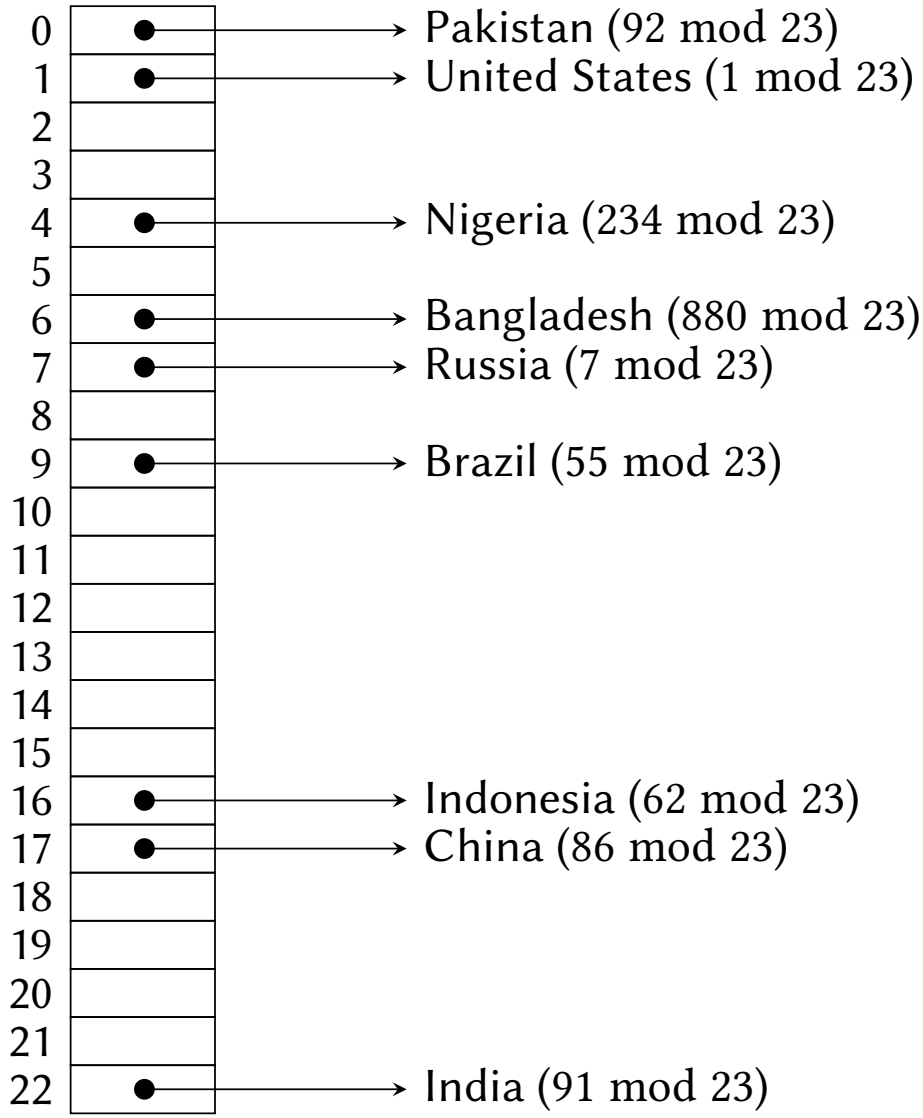
Inserting Nigeria.

Integer Hashing Example (8)



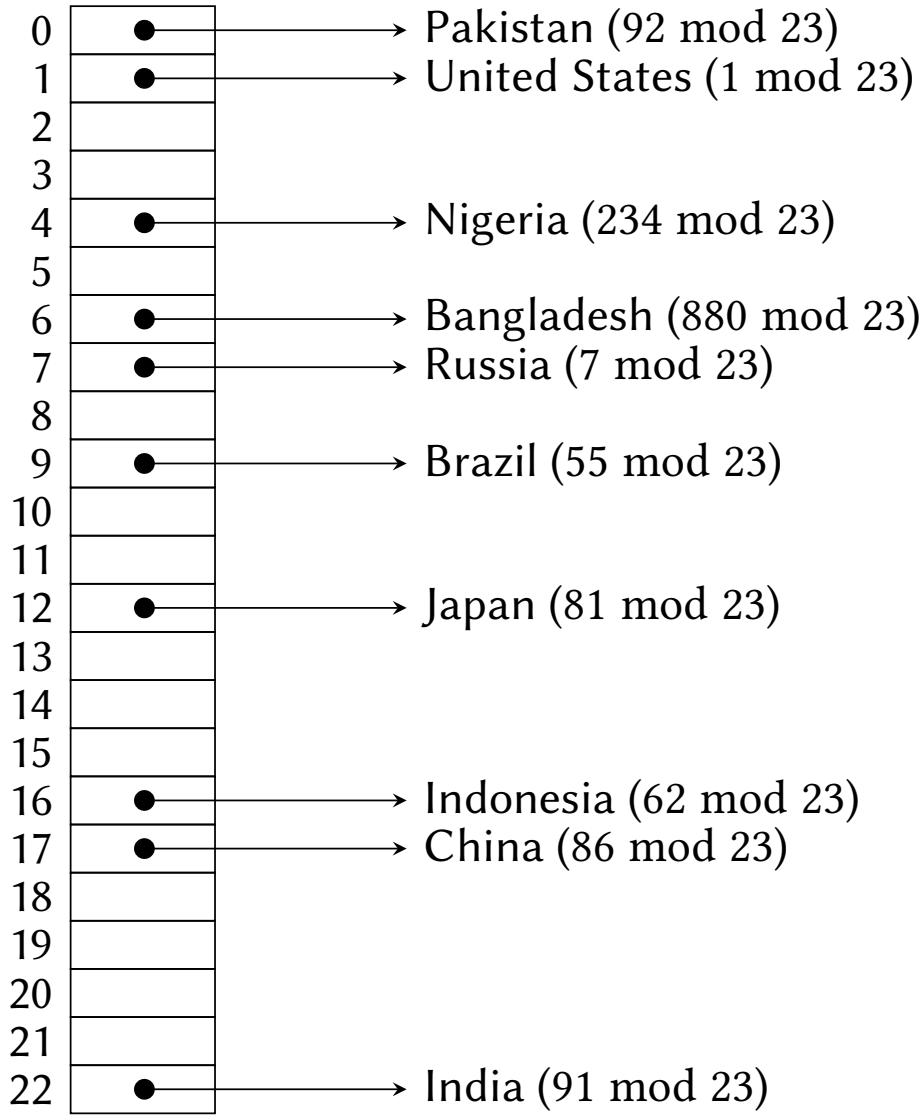
Inserting Bangladesh.

Integer Hashing Example (9)



Inserting Russia.

Integer Hashing Example (10)



Inserting Japan.

Requirements for the Hash Table

- The method works well, as it does not produce many collisions.
- This, however, depends on a good choice for the size of the hash table.
- For example, if the size of the hash table is 10^x, we will have many collisions.
- That is because the remainder of the division of a positive integer by 10^x is just the x last digits of the number.
- So all numbers with the same x last numbers will have the same hash value.

Hash Table Size 10^x

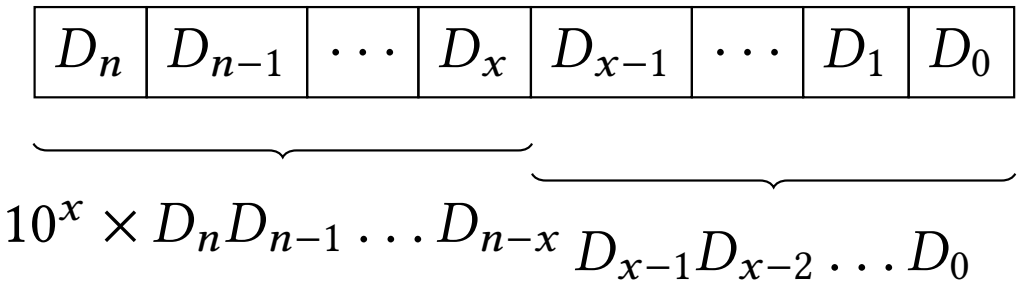
- For example, $12345 \bmod 100 = 45$, $2345 \bmod 100 = 45$, etc.
- In general a positive integer n with digits $D_n D_{n-1} \ldots D_1 D_0$ has value $D_n \times 10^n + D_{n-1} \times 10^{n-1} + \cdots + D_1 \times 10^1 + D_0 \times 10^0$.
- For each power of 10, 10^x, we have:

$$\frac{D_n D_{n-1} \ldots D_1 D_0}{10^x} = 10^x \times D_n D_{n-1} \ldots D_x + x_{x-1} D_{x-2} \ldots D_1 D_0$$

- Therefore it is:

$$D_n D_{n-1} \ldots D_1 D_0 \bmod 10^x = D_{x-1} D_{x-2} \ldots D_1 D_0$$

Hash Table Size 10^x (Contd.)



Remainder of division by a power 10^x.

Hash Table Size in Other Number Systems

- Similar problems occur in other number systems.
- If we use a number system with base b, then if the size of the hash table is b^x, only the x last digits will matter when calculating the hash value.

Uniform Distribution

- Ideally, we would like to have a hash function that distributes the keys to the table with equal probability for each position.
- In other words, the keys distribution must be *uniform*.
- A probability distribution is called uniform if all values have the same probability.
- This guides as away from some choices regarding the size of the hash table.
- If it is an even number, for instance, all even keys will go to the even positions of the table and all odd keys to the odd positions.

Selecting the Hash Table Size

- In practice, a good choice is to choose a prime number for the size of the table.
- So, if we want to store about 1000 keys, we will use a table of size 997, which is a prime number, instead of an array of size 1000.

String Hashing

- If our keys are strings, we can handle them as if they were numbers written in a system with a suitable base, e.g., 26.
- If s is a string of n characters, its value will be:

$$v = \texttt{Ordinal}(s[0])b^{n-1} + \texttt{Ordinal}(s[1])b^{n-2} + \cdots + \texttt{Ordinal}(s[n-1])b^0$$

- The `Ordinal` function returns the position of the letter in the alphabet and b is the base of the number system.
- At the end we calculate the hash as:

$$h = v \bmod m$$

String Hashing Algorithm

Algorithm: A string hash function.

`StringHash`$(s, b, m) \rightarrow h$

Input: s, a string
b, the base of the number system
m, the size of the hash table

Output: h, the hash value of s

1 $v \leftarrow 0$
2 $n \leftarrow |s|$
3 **for** $i \leftarrow 0$ **to** n **do**
4 $\quad v \leftarrow v + \texttt{Ordinal}(s[i]) \cdot b^{n-1-i}$
5 $h \leftarrow v \bmod m$
6 **return** h

String Hashing Example

$$v_0 = \texttt{Ordinal}(\text{'H'}) \cdot 26^4 = 7 \cdot 456{,}976 = 3{,}198{,}832$$
$$v_1 = 3{,}198{,}832 + \texttt{Ordinal}(\text{'E'}) \cdot 26^3 = 3{,}198{,}832 + 4 \cdot 26^3 = 3{,}269{,}136$$
$$v_2 = 3{,}269{,}136 + \texttt{Ordinal}(\text{'L'}) \cdot 26^2 = 3{,}269{,}136 + 11 \cdot 26^2 = 3{,}276{,}572$$
$$v_3 = 3{,}276{,}572 + \texttt{Ordinal}(\text{'L'}) \cdot 26^1 = 3{,}276{,}572 + 11 \cdot 26 = 3{,}276{,}858$$
$$v_4 = 3{,}276{,}858 + \texttt{Ordinal}(\text{'O'}) = +3{,}276{,}858 + 14 = 3{,}276{,}872$$
$$h = 3{,}276{,}872 \bmod 31 = 17$$

Hashing the string "HELLO".

Correspondence between String Hashing and Polynomials

- The string hashing function corresponds to the calculation of a polynomial.
- Specifically, we find the remainder of the division by m of the polynomial:

$$p(x) = a_{n-1}x^{n-1} + a_{n-2}x^{n-2} + \cdots + a_0$$

- The coefficients of the polynomial are the characters of the string and the evaluation is done for $x = b$.
- For "HELLO", the polynomial is:

$$p(x) = 7x^4 + 4x^3 + 11x^2 + 11x + 14$$

which we calculate for $x = 26$.

Complexity of Polynomial Evaluation

- The way we worked, to evaluate the n degree polynomial $p(x) = a_n x^n + a_{n-1} x^{n-1} + \cdots + a_0$, we calculated all the powers from the left to the right.
- To evaluate $a_n x^n$ we need $n - 1$ multiplications (to calculate the power) plus a multiplication with a_n, so n multiplications in total.
- To evaluate $a_{n-1} x^{n-1}$ we need $n - 1$ multiplications.
- In total we need $n + (n-1) + \cdots + 1 = n(n-1)/2$ multiplications and n additions to add the terms.

Horner's Rule

- A more efficient way to evaluate a polynomial is *Horner's rule*.
- The idea is to rearrange the polynomial as follows:

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n = \Big(\ldots (a_n x + a_{n-1}) x + \cdots \Big) x + a_0$$

- In this arrangement, we evaluate from the inside to the outside.

Example of Horner's Rule

$$\Big(\ldots(a_n x + a_{n-1})x + \cdots\Big)x + a_0$$

$$r_n$$

$$\Big(\ldots(r_n x + a_{n-1})x + \cdots\Big)x + a_0$$

$$r_{n-1}$$

$$\Big(\ldots(r_{n-1} x + a_{n-2})x + \cdots\Big)x + a_0$$

$$\vdots$$

$$(b_2 x + a_1)x + a_0$$

$$r_1$$

$$r_1 x + a_0$$

$$r_0$$

$$\Big(\big((7x + 4)x + 11\big)x + 11\Big)x + 14$$

$$\big((186x + 11)x + 11\big)x + 14$$

$$(4{,}847x + 11)x + 14$$

$$126{,}033x + 14$$

$$3{,}276{,}872$$

Where $x = 26$.

Horner's Rule Algorithm

Algorithm: Horner's rule.

`HornerRule`$(A, x) \rightarrow r$

Input: A, an array containing the coefficients of a polynomial of degree n
x, the point at which to evaluate the polynomial

Output: r, the value of the polynomial at x

```
1  r ← 0
2  foreach c in A do
3      r ← r · x + c
4  return r
```

Horner's Rule Complexity

- Line 3 of the algorithm is executed n times.
- We therefore need n multiplications and n additions.

Improving String Hashing

In $a \bmod b$ the following properties hold:

$$(a + b) \bmod m = \big((a \bmod m) + (b \bmod m)\big) \bmod m$$

$$(ab) \bmod m = \big((a \bmod m) \cdot (b \bmod m)\big) \bmod m$$

That means we can avoid calculating big powers by taking the remainder of m as we go.

Optimized String Hashing

Algorithm: An optimized string hash function.

`OptimizedStringHash`$(s, b, m) \rightarrow h$

Input: s, a string
b, the base of the number system
m, the size of the hash table

Output: h, the hash value of s

1 $h \leftarrow 0$
2 **foreach** c **in** s **do**
3 $h \leftarrow \left(b \cdot h + \texttt{Ordinal}(c)\right) \bmod m$
4 **return** h

Example of Optimized String Hashing

$$h_0 = \texttt{Ordinal}(\text{`H'}) \bmod 31 = 7 \bmod 31 = 7$$

$$h_1 = \big(26 \cdot 7 + \texttt{Ordinal}(\text{`E'})\big) \bmod 31 = (182 + 4) \bmod 31 = 0$$

$$h_2 = \big(26 \cdot 0 + \texttt{Ordinal}(\text{`L'})\big) \bmod 31 = 11 \bmod 31 = 11$$

$$h_3 = \big(26 \cdot 11 + \texttt{Ordinal}(\text{`L'})\big) \bmod 31 = (286 + 11) \bmod 31$$

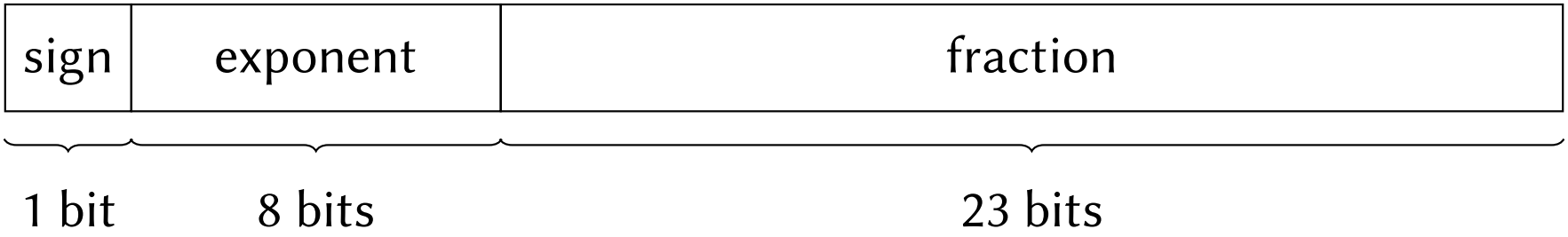
$$h_4 = \big(26 \cdot 18 + \texttt{Ordinal}(\text{`O'})\big) \bmod 31 = 482 \bmod 31 = 17$$

Outline

Floating Point Hashing

- Apart from integers and strings, our keys can of course be floating point numbers.
- What kind of hash function can we use for floating point numbers?
- One idea would be to convert them to a string and then work with that. For example, we could convert the number number 261.63 to the string “261.63”.
- But that is a slow process.
- Another idea would be to take out the decimal point and get the corresponding integer: from 261.63, get 26163.
- However, that is not easy to do because floating point numbers are not represented in computers the way they look.

Floating Point Representation



Floating point representation (for 32 bits, a similar way is used for 64 bits). The value of a number is:

$$(-1)^s \times 1.f \times 2^{e-127}$$

where s is the sign, f is the fraction, also called *characteristic*, *mantissa*, or *significant*, and e is the exponent.

Binary Fractional Numbers

- If a binary number has an integer and a fractional part, its value is the sum of the two parts.
- Therefore the number $B_1B_0, B_1B_2 \ldots B_n$ has value $B_1 \times 2^2 + B_0 + B_1 \times 2^{-1} + B_2 \times 2^{-2} + \cdots + B_n \times 2^{-n}$.
- For example, number 1.01 in binary is equal to $1 + 0 \times 2^{-1} + 1 \times 2^{-2} = 1.25$ in decimal.

Floating Point Representation Examples

0	01110011	00000110001001001101111

$$(-1)^0 \times 2^{115-127} \times 1{,}02400004864 = 0.00025$$

1	10010101	10001101010101101100000

$$(-1)^1 \times 2^{149-127} \times 1.55210494995 = -6{,}510{,}000$$

0	10000111	00000101101000010100100

$$(-1)^0 \times 2^{135-127} \times 1.02199220657 = 261.63$$

Floating Point Hashing

- In order to hash a floating point, we interpret its bits as if it were an integer.
- Then we hash that integer.
- For example, 261.63 is represented as 01000011100000101101000010100100.
- Taking it as an integer, we have

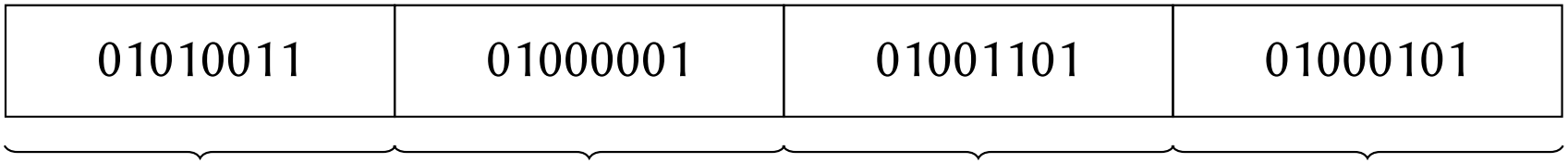
$$(01000011100000101101000010100100)_2 = (1132646564)_{10}$$

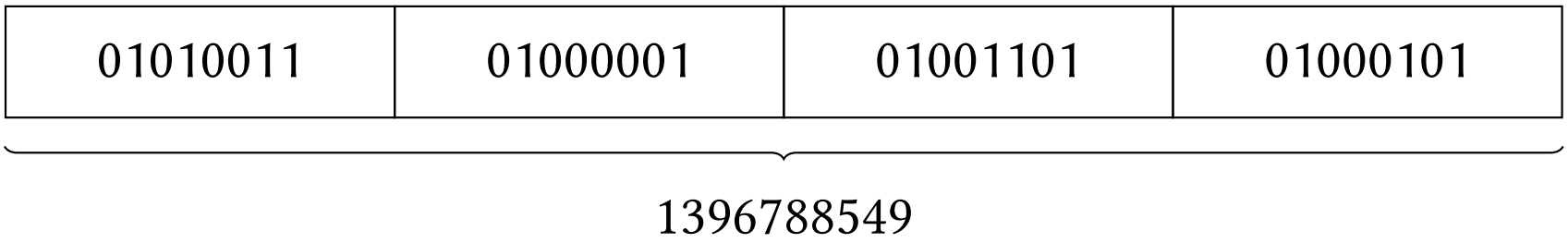
- So we use the number 1132646564 as input to the string hashing function.

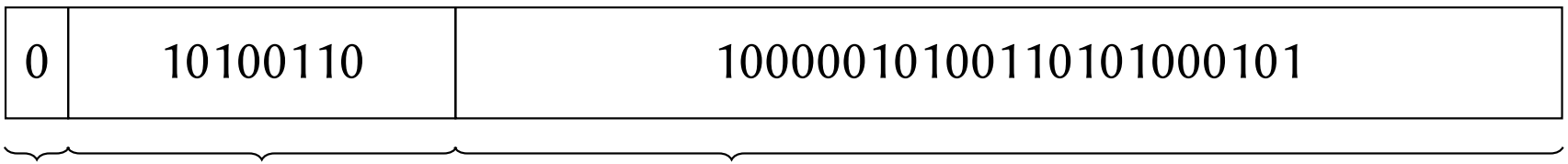
Bits Interpretation

- Be careful! The computer has no idea what a sequence of bits stands for.
- The same sequence of bits may represent a string, an integer, or a floating point number.
- It is up to us, when we write our programs, to make sure we read our data with the intended interpretation.

Bits Interpretation Examples





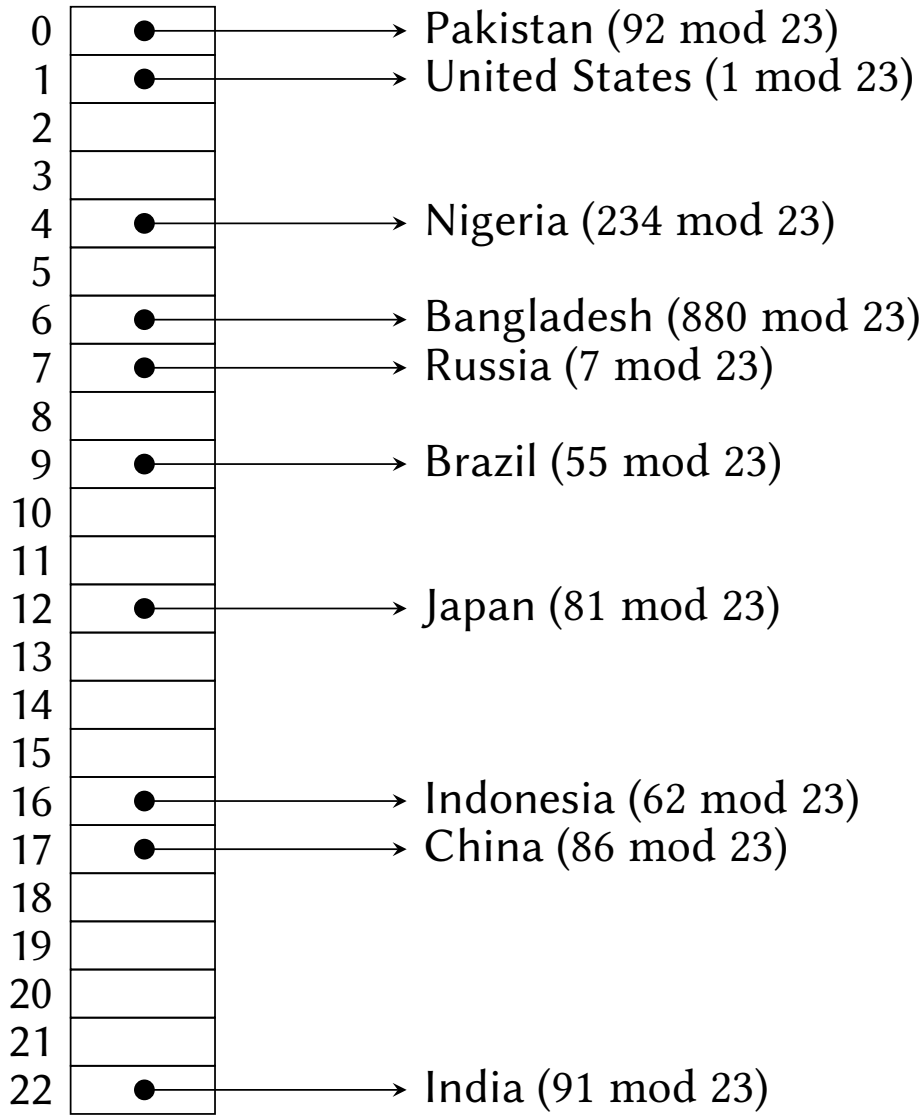


$$(-1)^0 \times 2^{166-127} \times 1.51017057896 = 8.30225055744 \times 10^{11}$$

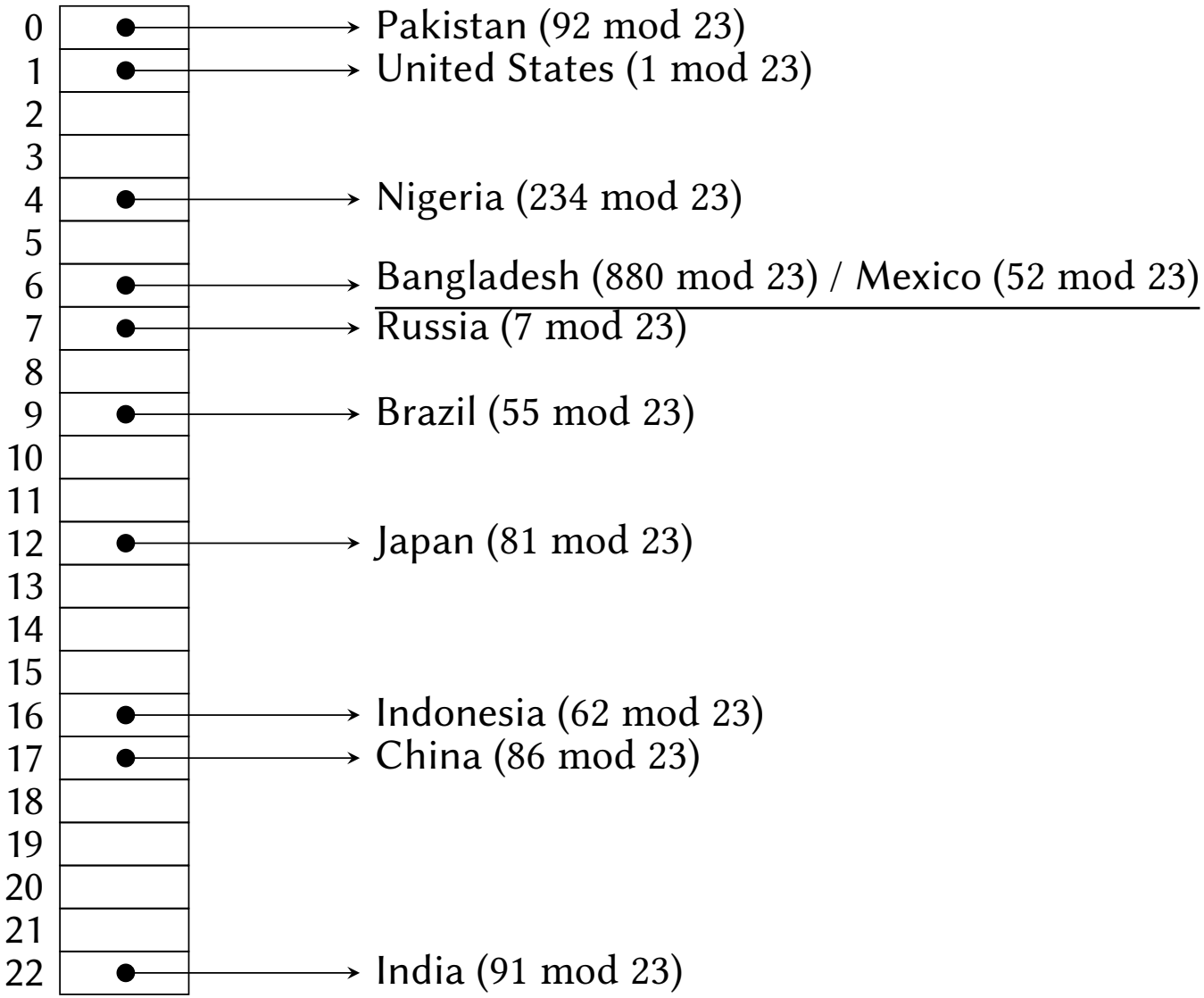
Outline

- Because of the pigeonhole principle, we will always have collisions.
- The best we can do is to reduce them as much as possible, by using a good hash function and a good size for the hash table.
- Nevertheless, we must have a way to handle them when they occur.

Collisions Example (1)



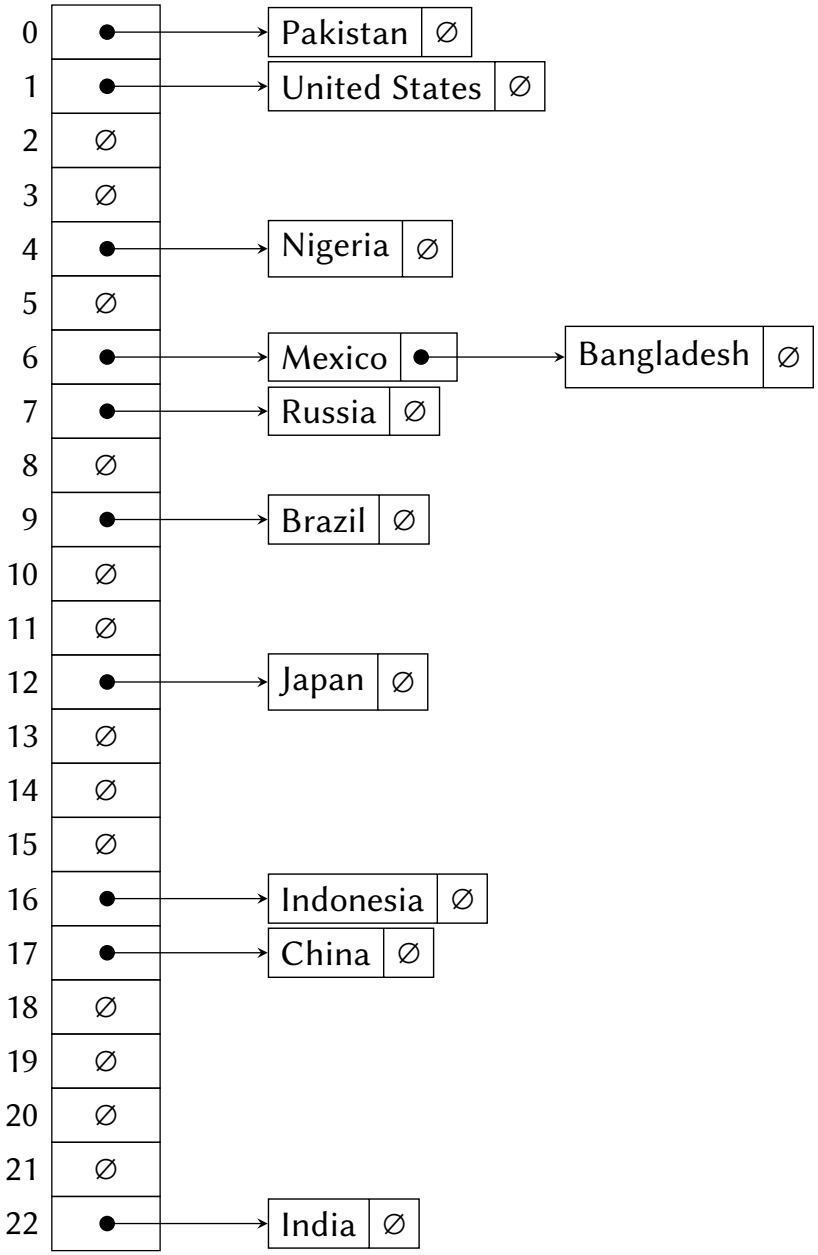
Collisions Example (2)



Chains

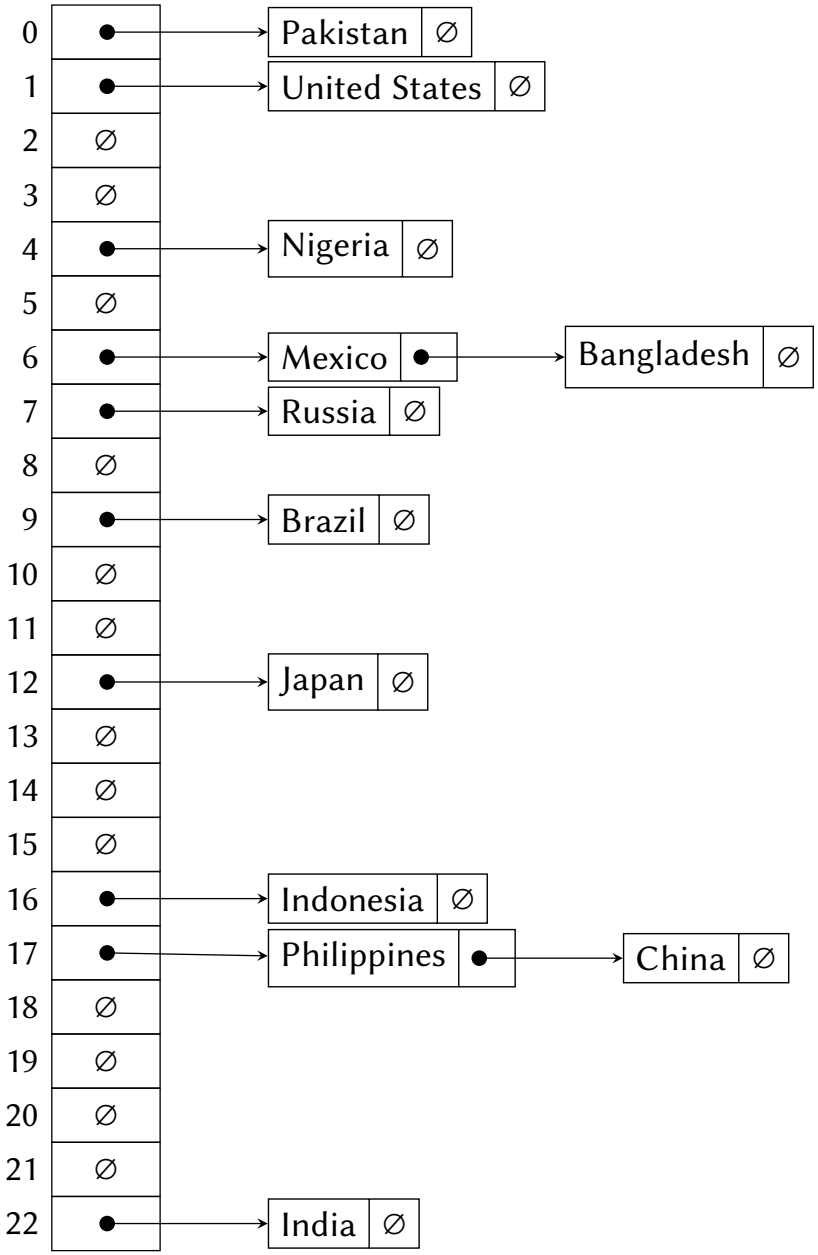
- The most common way to handle collisions is to store in the hash table lists of records.
- Each bucket will contain a list with all the records whose keys conflict together.
- That explains the name “bucket”.
- If there are no keys for a bucket, it will point to NULL.
- We call the resulting lists *chains*.
- This method is called *hashing with separate chaining*.

Collisions Handling Example (1)



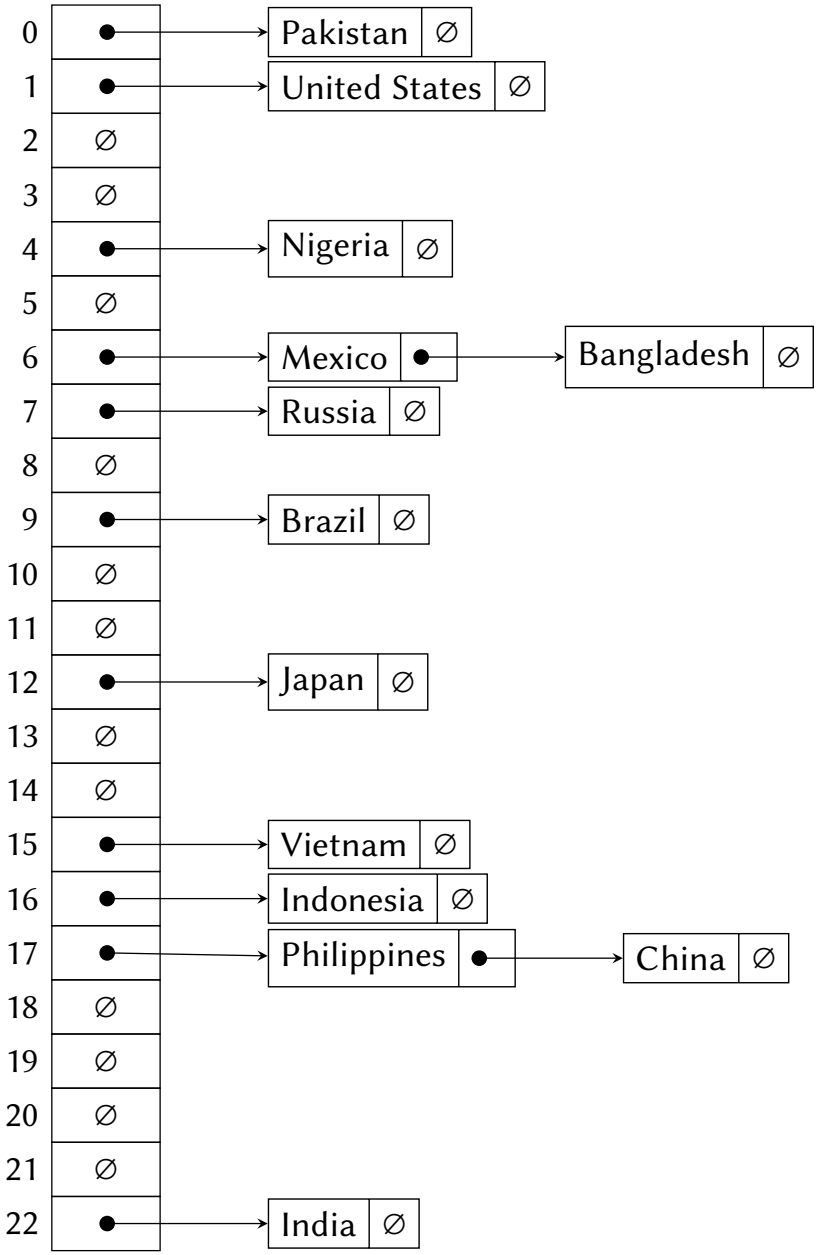
Inserting Mexico (collision).

Collisions Handling Example (2)



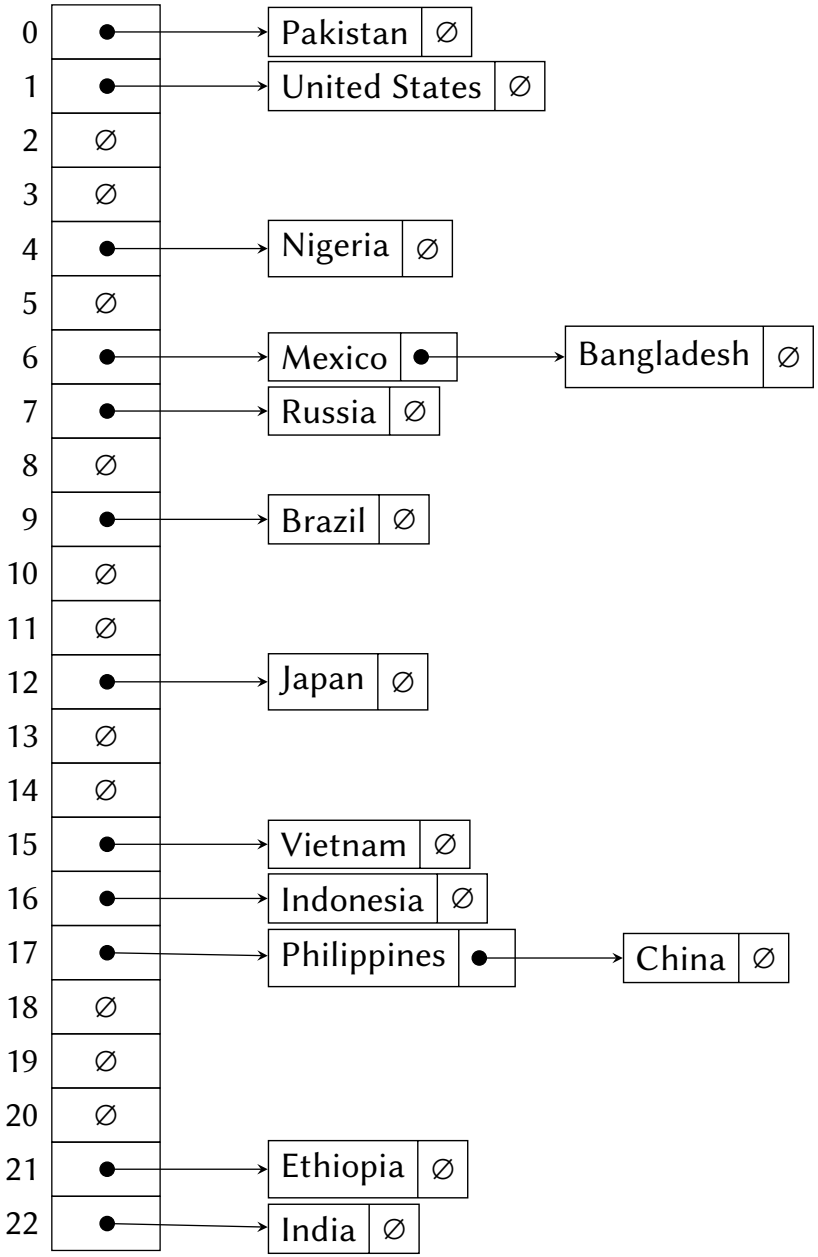
Inserting Philippines (collision).

Collisions Handling Example (3)



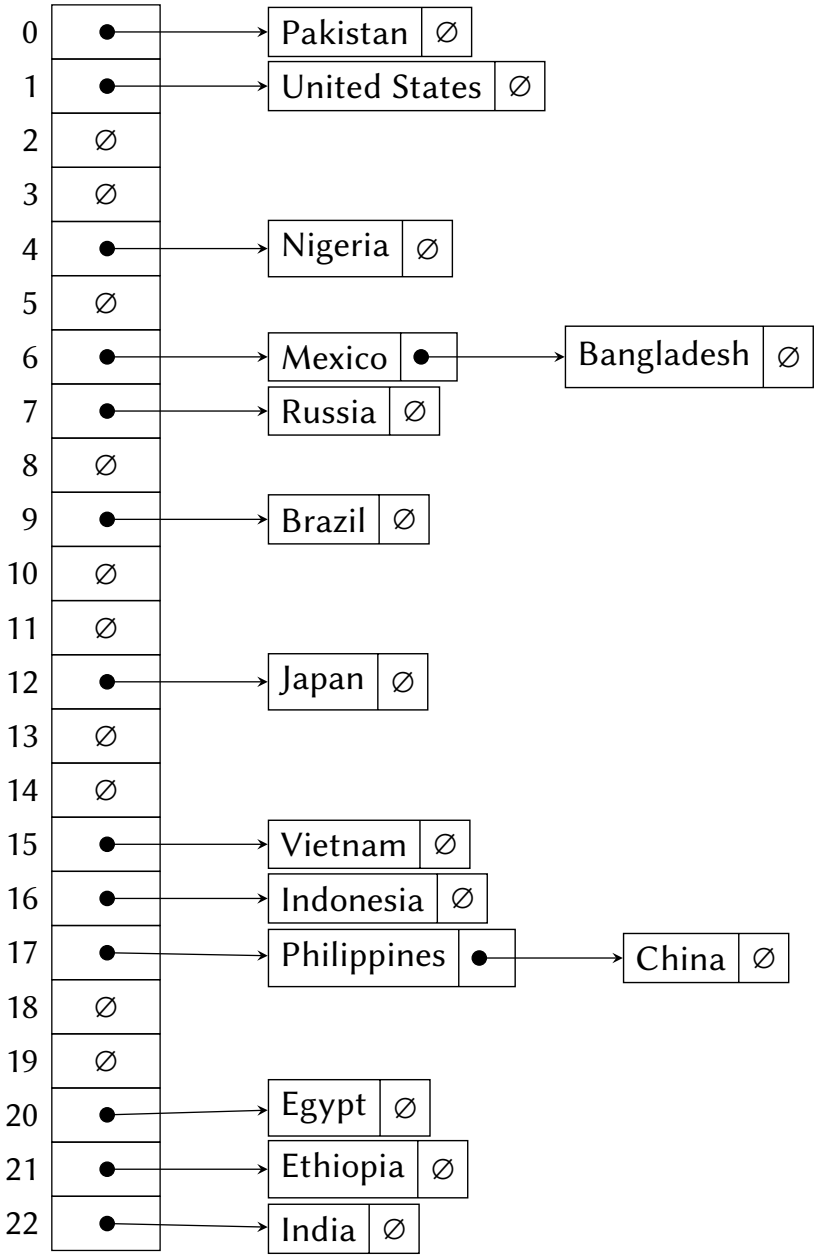
Inserting Vietnam.

Collisions Handling Example (4)



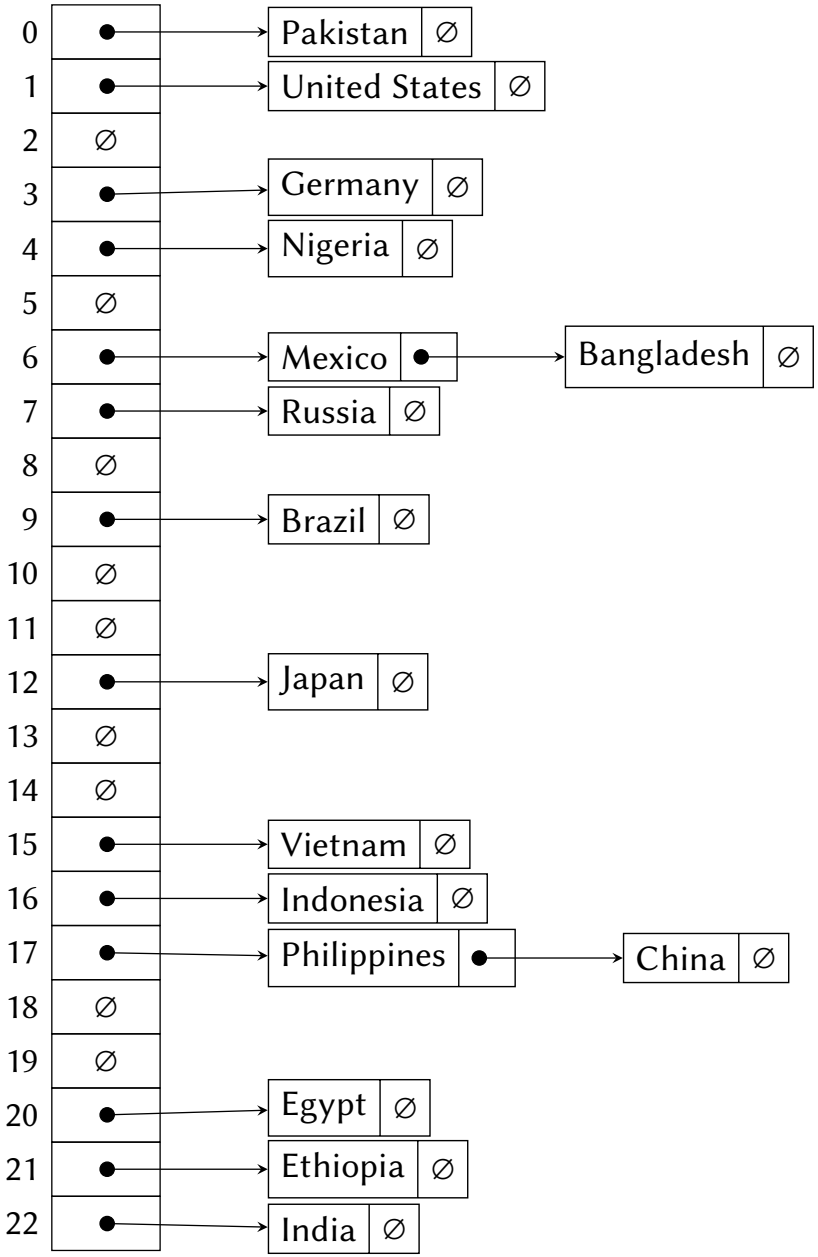
Inserting Ethiopia.

Collisions Handling Example (5)



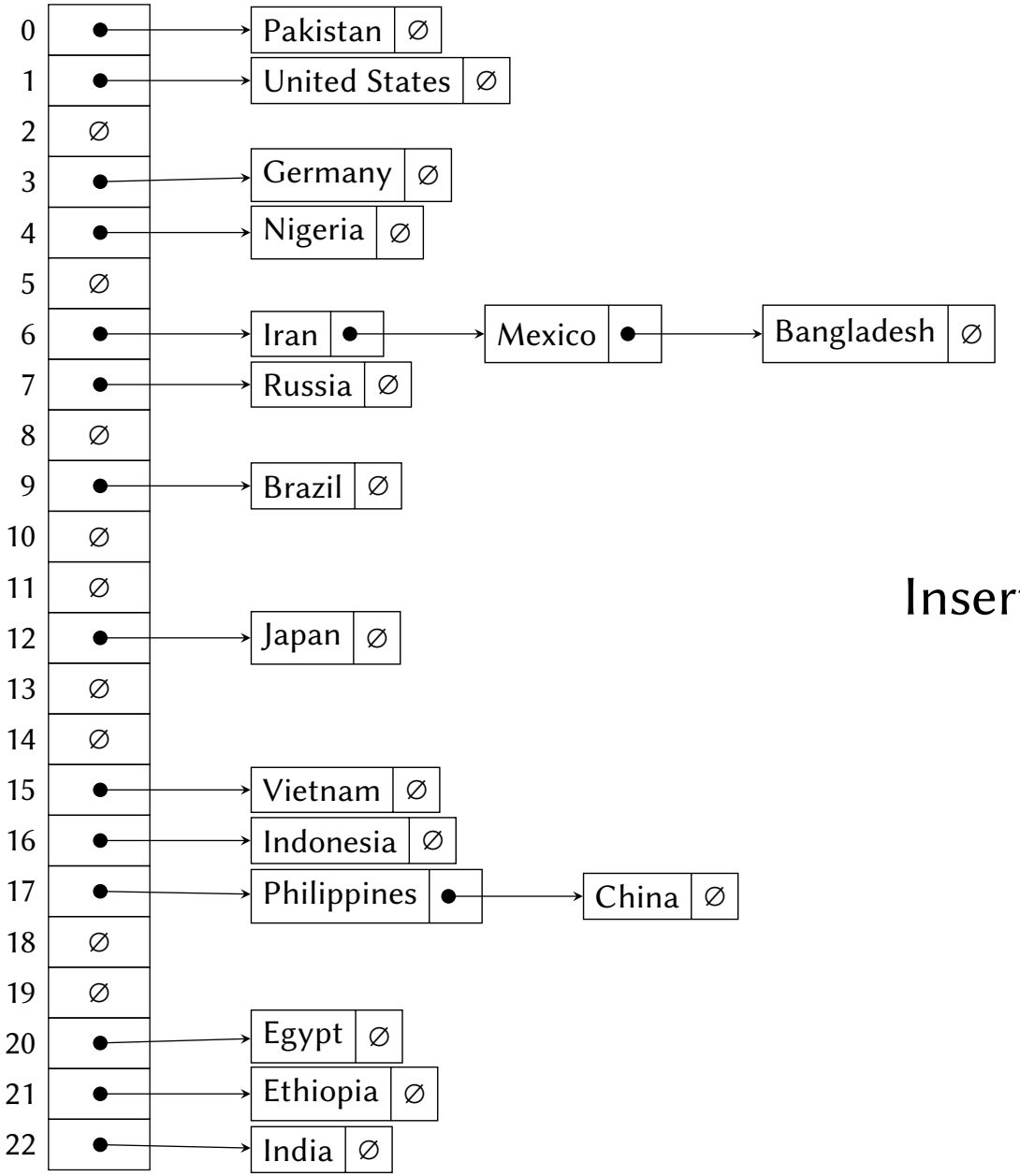
Inserting Egypt.

Collisions Handling Example (6)



Inserting Germany.

Collisions Handling Example (7)



Inserting Iran (collision).

Insertion in Hash Table

Algorithm: Insertion in hash table with chained lists.

`InsertInHash`(T, x)

Input: T, a hash table

x, a record to insert in the hash table

Result: x is inserted into T

1 $h \leftarrow$ `Hash(Key(`x`))`

2 `InsertInList(`$T[h]$, NULL, x`)`

The function `InsertInList`(L, p, x) inserts a new node with data x in list L after node p. If p is NULL, the new node is inserted at the beginning of the list.

Search in Hash Table

Algorithm: Search in hash table with chained lists.

SearchInHash(T, x) $\longrightarrow$ TRUE or FALSE

Input: T, a hash table

x, a record to lookup in the hash table

Output: TRUE if found, FALSE otherwise

```
1  h ← Hash(Key(x))
2  if SearchInList(T[h], x) = NULL then
3      return FALSE
4  else
5      return TRUE
```

The function SearchInList(L, x) searches for element x in L and returns it, if found, or NULL otherwise.

Remove from Hash Table

Algorithm: Removal from hash table with chained lists.

RemoveFromHash(T, x) $\longrightarrow$ x or NULL

Input: T, a hash table

x, a record to remove from the hash table

Output: x, the record x if it was removed, or NULL if x was not in the hash table

```
1  h ← Hash(Key(x))
2  return RemoveFromList(T[h], x)
```

The RemoveFromList(L, x) function removes element x from L and returns it, if found; otherwise it returns NULL.

Search Performance in Hash Tables (1)

- First of all, we must calculate the hash value.
- If our keys are numeric, this is the cost of a division, which we take to be constant, $O(1)$.
- If the keys are strings, the cost is $\Theta(n)$, where n is the length of the string. Because the hash function is very fast and it depends on the key and not the number of records, we usually consider it constant as well.

Search Performance in Hash Tables (2)

- To that we must add the cost for searching for the key.
- If the bucket is empty, then we have one comparison, so it is $O(1)$.
- Otherwise, the cost is $O(|L|)$, where $|L|$ is the length of the list.
- So the question is, how long can the chain be?

Search Performance in Hash Tables (3)

- The answer depends on the hash function.
- If it is good, then the length of each chain will be about n/m, where n is the number of keys and m is the size of the hash table.
- The number n/m is called *load factor* of the hash table.
- For an unsuccessful search we need time equal to $\Theta(n/m)$ to reach the end of the chain.
- For a successful search it can be proved that the time we need is $\Theta(1 + (n-1)/2m)$.

Search Performance in Hash Tables (4)

- In both successful and unsuccessful searches, the time required depends on n/m.
- So, if the number of the keys is proportional to the size of the hash table, i.e., $n = cm$, the search time is constant.
- Indeed, for an unsuccessful search we have $\Theta(n/m) = \Theta(cm/m) = \Theta(c) = O(1)$. A successful search is faster, so again we have $O(1)$.

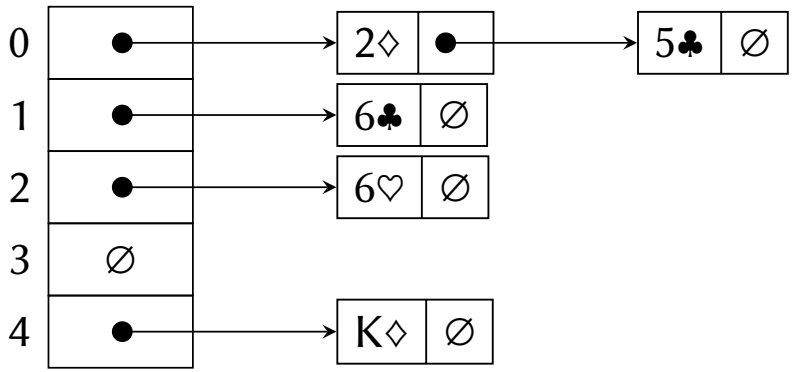
Search Performance in Hash Tables (5)

- If $n = cm$ the search time cannot be greater than c; therefore we want to ensure that c remains small.
- For example, if $n = 2m$, then searches will not need more than two comparisons.
- So, if we know how many items we will insert in the hash table, we can select the right size for the hash table.

Hash Table Resizing

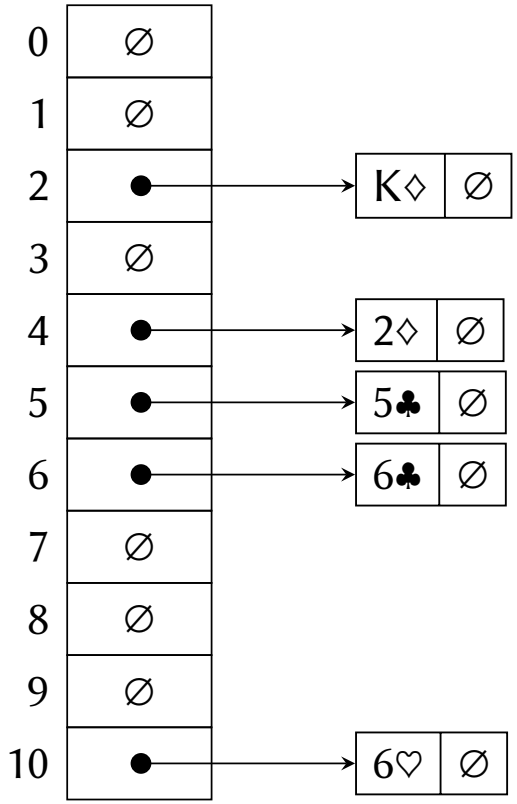
- If we do not know in advance how many items we will store in the hash table, or if we underestimate, we can set a limit to the load factor.
- When the load factor reaches the limit, we create a new, bigger, hash table, for example double the previous one.
- We insert all the elements of the existing table to the new table.
- We delete the first table.
- In this way we can guarantee that searches will still happen at constant time, but insertion will be slow when we need to resize the table.

Resizing Example (1)



The load factor is 1.

Resizing Example (2)



The load factor is about 0.45.

More on Performance

- The performance of hash tables is probabilistic.
- We expect that on average the length of the chains will not exceed m/n, but this may happen.
- Moreover, resizing requires time, when it happens (rarely).

More on Performance (Contd.)

- At the same time, we can trade-off time with space.
- If time is more important, we can use more space to have a lower load factor.
- If space is more important, we can stuff more keys in the hash table.

Outline

Uses of Hash Tables

- Hash tables are very popular.
- They are used in all kinds of applications.
- They are also the foundation for creating data structures such as *sets* and *dictionaries*.

Sets

- A *set* is a data structure containing items, each one of them unique.
- A set can be implemented directly with a hash table.
- Inserting an item into the set is the same as inserting into the hash table.
- Removing an item from the set is the same as removing it from the hash table.
- Checking membership in a set is the same as looking for an item in the hash table.
- Sets, as in mathematics, are not ordered.

Dictionaries

- Another data structure that is implemented with hash tables are *dictionaries*.
- They are also called *maps* or *associative arrays*.
- A dictionary contains key-value pairs.
- When we look up a key, we get the corresponding value (as happens when we look up a word in a dictionary).
- We can insert key-value pairs, remove key-value pairs, or change the value associated with a key.
- In contrast to real-world dictionaries, these dictionaries are not ordered.

Insertion in a Dictionary

Algorithm: Insertion in a dictionary (map).

`InsertInMap`(T, k, v)

Input: T, a hash table
k, the key of the key-value pair
v, the value of the key-value pair

Result: value v is inserted into the dictionary associated with the key k

```
1  h ← Hash(k)
2  p ← SearchInListByKey(T[h], k)
3  if p = NULL then
4      p ← CreateArray(2)
5      p[0] ← k
6      p[1] ← v
7      InsertInList(T[h], NULL, p)
8  else
9      p[1] = v
```

Look up in a Dictionary

Algorithm: Lookup in a dictionary (map).

Lookup$(T, k) \longrightarrow v$ or NULL

Input: T, a hash table
k, a key

Output: v, the corresponding value, if it exists, or NULL otherwise

```
1  h ← Hash(k)
2  p ← SearchInHash(T[h], k)
3  if p = NULL then
4      return NULL;
5  else
6      return p[1]
```

Removal from Dictionary

Algorithm: Removal from dictionary (map).

`RemoveFromMap`$(T, k) \longrightarrow [k, v]$

Input: T, a hash table

k, a key to remove the corresponding key-value pair from the dictionary

Output: $[k, v]$, the key-value pair corresponding to k if it was removed, or NULL if no corresponding key-value pair was found in the dictionary

```
1  h ← Hash(k)
2  return RemoveFromListByKey(T[h], k)
```

Outline

Fingerprints

- A fingerprint is a tiny piece with which we can identify a person.
- Can we use something similar to identify digital artifacts?
- For example, can we identify a song?

Digital Fingerprints

- A *digital fingerprint* is a fingerprint of a digital artifact.
- It is a small piece, derived from the original artifact, by which we can identify the artifact.
- As normal fingerprints, we must be able to extract it and match it easily.

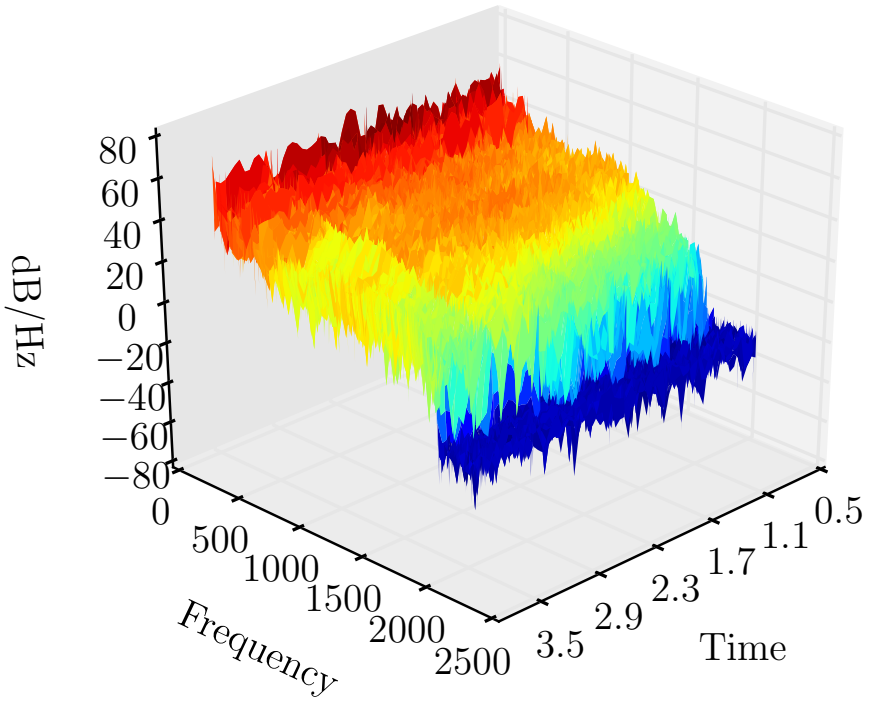
Identifying Songs

- We start with a song, or a piece of a song.
- We calculate its digital fingerprint.
- We have stored the digital fingerprints of known songs.
- To identify the song we are interested in, we only need to find a fingerprint from the known songs that matches the fingerprint of the song we are trying to identify.

Frequencies

- Sound is a vibration that passes through a medium (usually air).
- A sound has one or more frequencies associated with it.
- A pure note has a single frequency.
- For example, the note A has a frequency 440 Herz (Hz) in an even tempered scale and the note C has a frequency of 261.63Hz.
- A sound clip has many frequencies, changing over time, as different instruments and voices combine together.
- The intensity of each frequency corresponds to its energy.

Frequencies, Energies, and Time

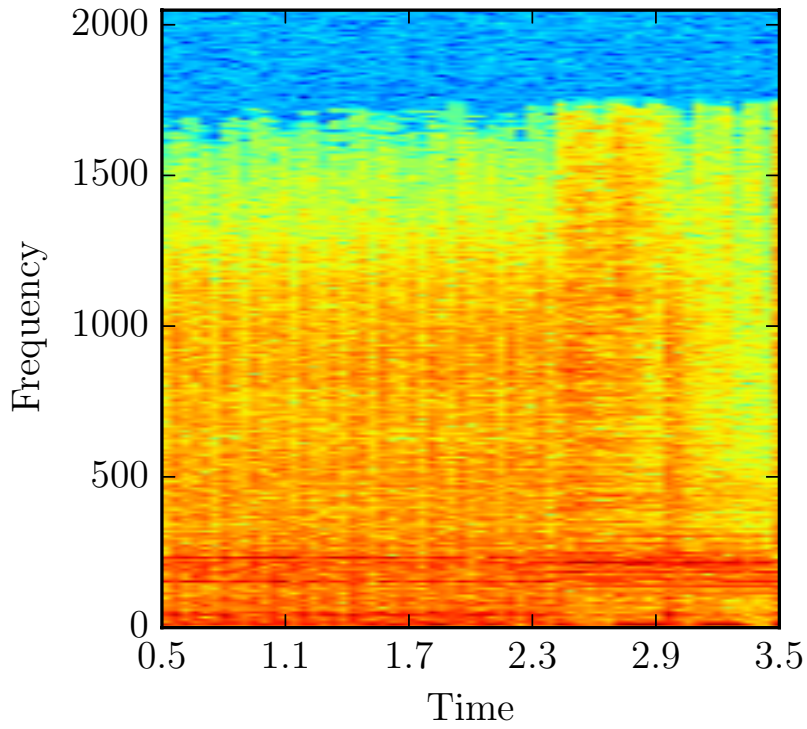


3D representation of energies, frequencies, and time.

Decibels

- In the z-axis we measure frequency energy as decibels (dB) by Herz.
- The negative sign comes into play because a decibel is logarithmic.
- It represents the ratio of two values: $10 \log(v/b)$, where b is the base value.
- In our case we have $b = 1$, so for $v < 1$ we have negative values.

Spectrogram

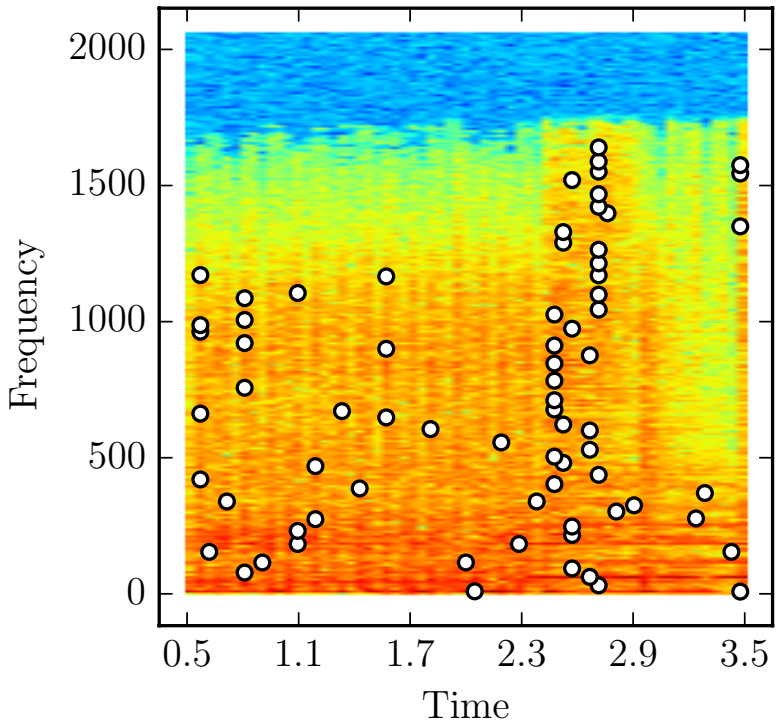


Frequencies, time, energy (red is highest energy).

Peaks

- The frequencies form a landscape in which there are peaks and troughs.
- We can detect peaks in a spectrogram mathematically.
- A peak is a point whose frequency is higher than that of its neighbors.

Spectrogram with Peaks



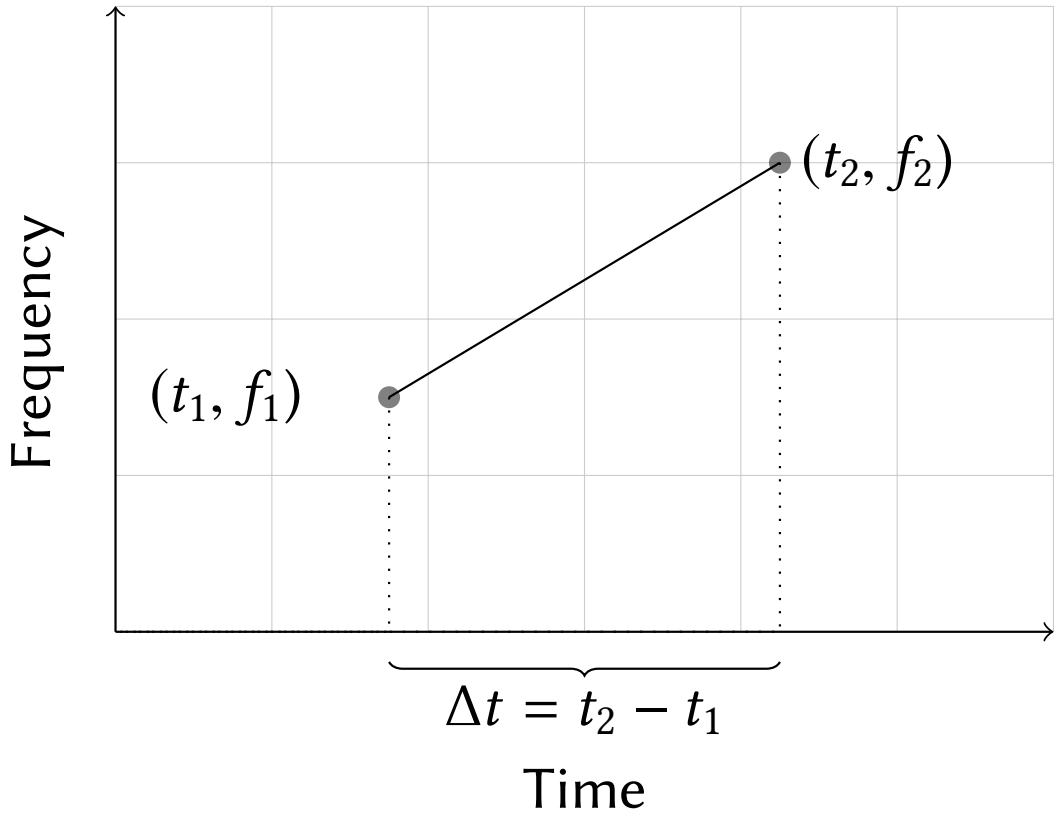
Peaks as Fingerprints

- The peaks of the frequencies could be used as the basis of a digital fingerprint.
- We could use the peaks as keys in a hash table. Then the records would be the song's data.
- We would then store into the hash table the keys and the records for all the songs we know. To identify a song, we look up in the hash table the peaks of the song.

Improvements

- In practice, using just the frequency peaks is not the best solution.
- It is better to use pairs of peaks and the time difference between them.
- If we have a peak f_1 at time t_1 and another peak f_2 at time t_2, instead of using peaks f_1 and f_2 as keys we create a new key $k = f_1 : f_2 : t_2 - t_1$.
- For example, such a key could be 1620.32:1828.78:350, where 350 is the milliseconds interval between the two peaks.

Hashing Frequency Peaks



The key is $f_1 : f_2 : \Delta t$.

Number of Keys

- The number of possible keys is very large.
- It is equal to the possible pairs of (f_x, f_y) in a song.
- That number is equal to the ways we can select k different elements out of n possible elements, where $k = 2$ and n the number of peaks in the song.

Permutations

- A *permutation* of k items out of n items is an ordered selection of k different items from the n items.
- There are n ways to select the first item, $n-1$ ways to select the second item, and so on, up to the kth element, for which we have only one choice.
- So in total we have:

$$n \times (n-1) \times \cdots \times (n-k+1)$$

- But:

$$n \times (n-1) \times \cdots (n-k+1) = \frac{n!}{(n-k)!}$$

Combinations

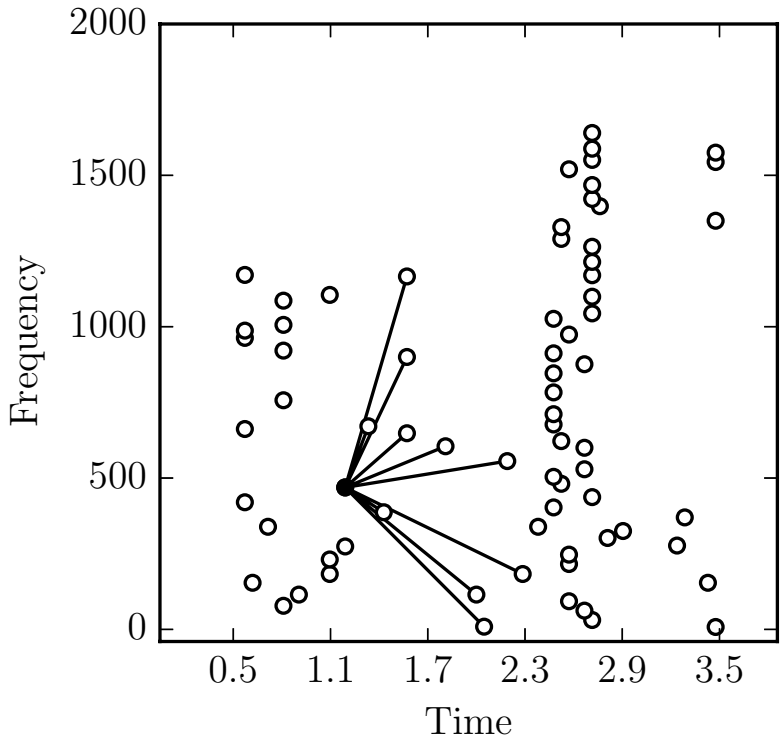
- A *combination* of k items out of n items is a selection without ordering of k items out of the n items.
- As in permutations order is important, in order to find the number of possible combinations, we need to divide the number of permutations by the number of different ways to order k items.
- There are k different possibilities for the first element, $k-1$ different possibilities for the second element, etc.
- Therefore we have $k \times (k-1) \times \cdots \times 1 = k!$ different ways to order k items.
- It follows that the number of possible combinations is:

$$\frac{n!}{k!(n-k)!} = \binom{n}{k}$$

Reducing the Number of Pairs

- The number $\binom{n}{2}$ can get very large.
- If we have 100 peaks in a song, we will have $\binom{n}{2} = 4950$ keys just for that song.
- To avoid an explosion in the size of the hash table, we do not use all possible pairs, but a pre-set number of pairs, e.g., up to ten pairs.
- This number is called *fan-out factor*.

Fan-out Factor Example



The fan-out factor is ten. We select the ten peaks that follow the peak we are examining.

Song Identification

- To populate the hash table we derive the spectrogram of each known song and use as keys the frequency pairings and their time difference based on a convenient fan-out factor.
- We derive the digital fingerprint of the song we are looking for.
- We look up in the hash table records with the same digital fingerprint.
- We pick the song with the most matches, as it is the one closest to the song we are looking for.

Outline

General

- Suppose we want to check whether some data exist in a set, without bothering to actually retrieve them.
- Suppose also that we do not care if for a small number of elements we may get *false positives*, i.e., a statement that they are in the set while they are not.
- Is there a way we can do that with better time and space characteristics than plain hash tables?

Bloom Filters

- An efficient way to check for set membership, with a low probability of false positives, is a *Bloom filter*.
- They were presented by Burton Bloom in 1970.
- They are called filters because they are frequently used to filter in only elements that belong to a known set of acceptable elements.

Basic Idea (1)

- We use a bit array T of size m.
- We also use k independent hash functions $h_0(x), h_1(x), \ldots h_{k-1}(x)$.
- We calculate the hash values for each element. Each hash function will return a value from zero to $m - 1$.
- To find k independent hash functions, we can use hash algorithms that take as input the key *and* an initialization parameter (a seed).

Basic Idea (2)

- If we find that $T[h_i(x)] = 1$ for all $i = 0, 1, \ldots k-1$, then we report that x is a member of the set. That may not be true, because of collisions, but we'll see that this happens rarely.
- If we find that $T[h_i(x)] \neq 1$ for any of $i = 0, 1, \ldots k-1$, then we can be sure that x is not in the set. We can insert it, if we want, by setting $T[h_i(x)] = 1$ for $i = 0, 1, \ldots k-1$.

Applications

- URLs shortening.
- Typeahead.
- Hyphenation.
- Spell checking.
- Web caching.
- Digital forensics.
- Bitcoin.
- ...

Bloom Filter Insertion

Algorithm: Insert in Bloom filter.

`InsertInBloomFilter`(T, x)

Input: T, a bit array of size m

x, a record to insert to the set represented by the Bloom filter

Result: the record is inserted in the Bloom filter by setting the bits $h_0(x), h_1(x), \ldots, h_{k-1}(x)$ to 1

```
1  for i ← 0 to k do
2      h ← h_i(x)
3      T[h] ← 1
```

Membership Check

Algorithm: Check membership with Bloom filter.

`IsInBloomFilter`$(T, x) \longrightarrow$ TRUE or FALSE

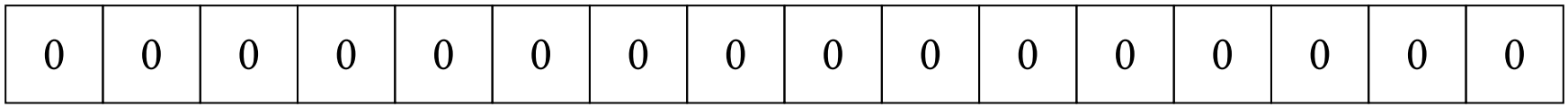
Input: T, a bit array of size m

x, a record to check if it is a member of the set represented by the Bloom filter

Output: TRUE if x is in the Bloom filter, FALSE otherwise

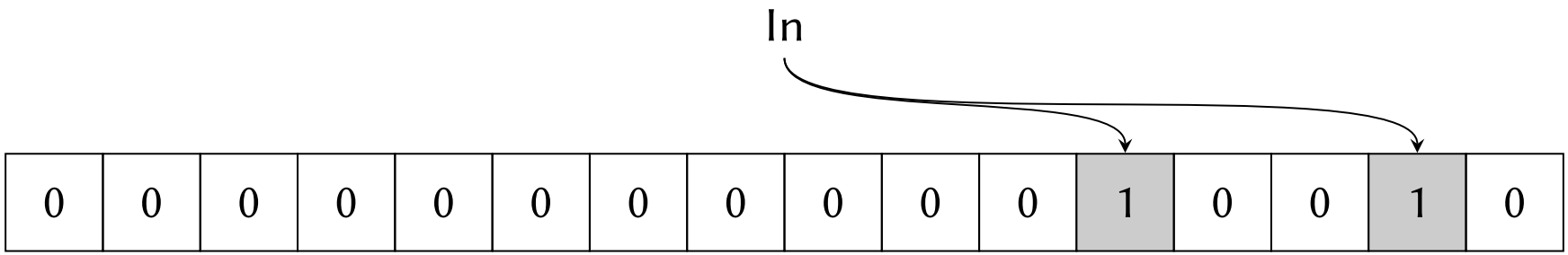
```
1  for i ← 0 to k do
2      h ← h_i(x)
3      if T[h] = 0 then
4          return FALSE
5  return TRUE
```

Bloom Filter Example (1)



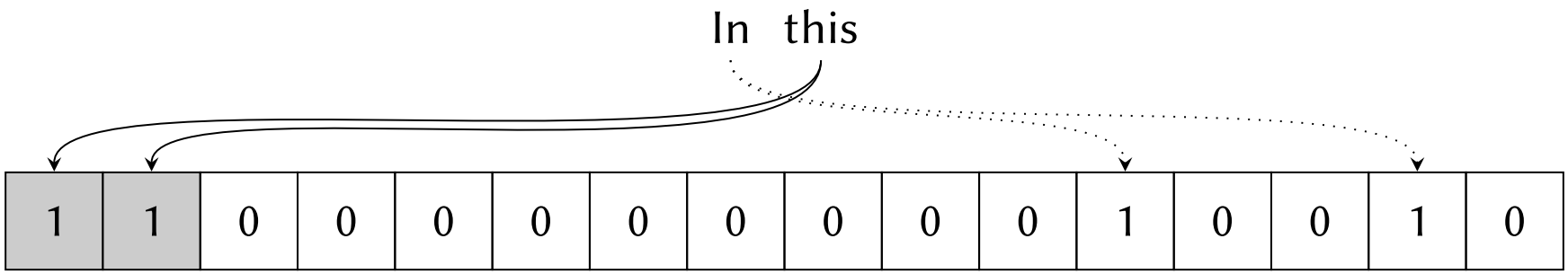
16 bit Bloom filter, initial state.

Bloom Filter Example (2)



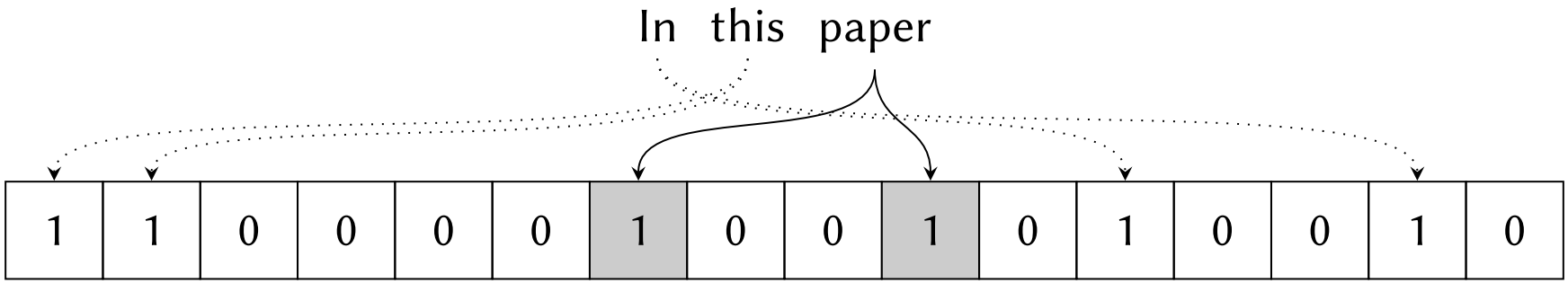
16 bit Bloom filter, inserting “In” (positions 14, 11).

Bloom Filter Example (3)



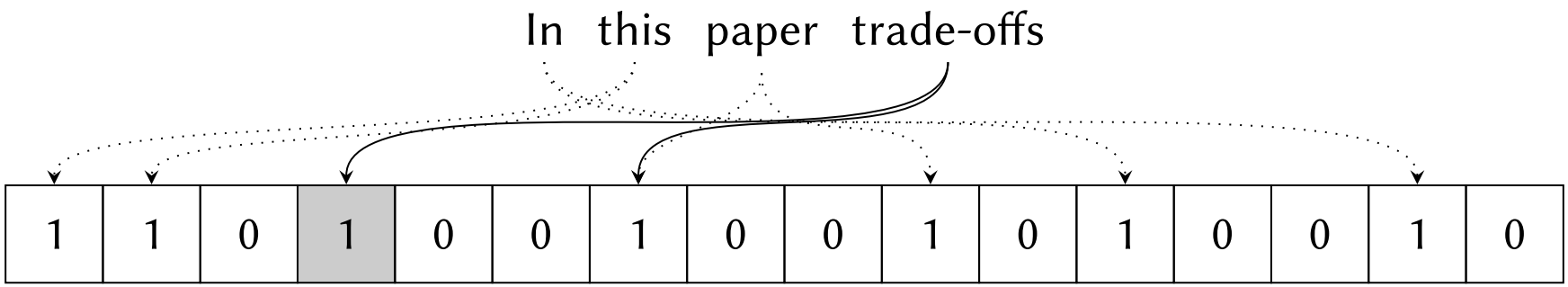
16 bit Bloom filter, inserting “this” (positions 1, 0).

Bloom Filter Example (4)



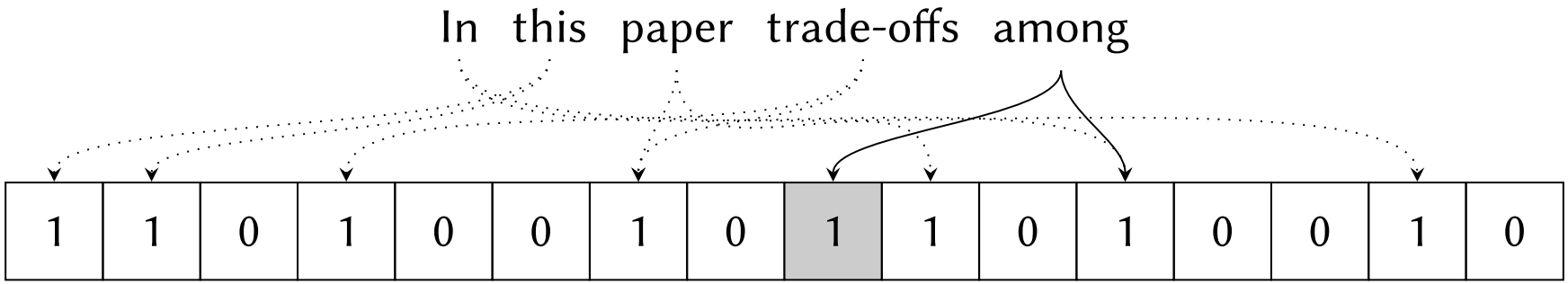
16 bit Bloom filter, inserting “paper” (position 9, 6).

Bloom Filter Example (5)



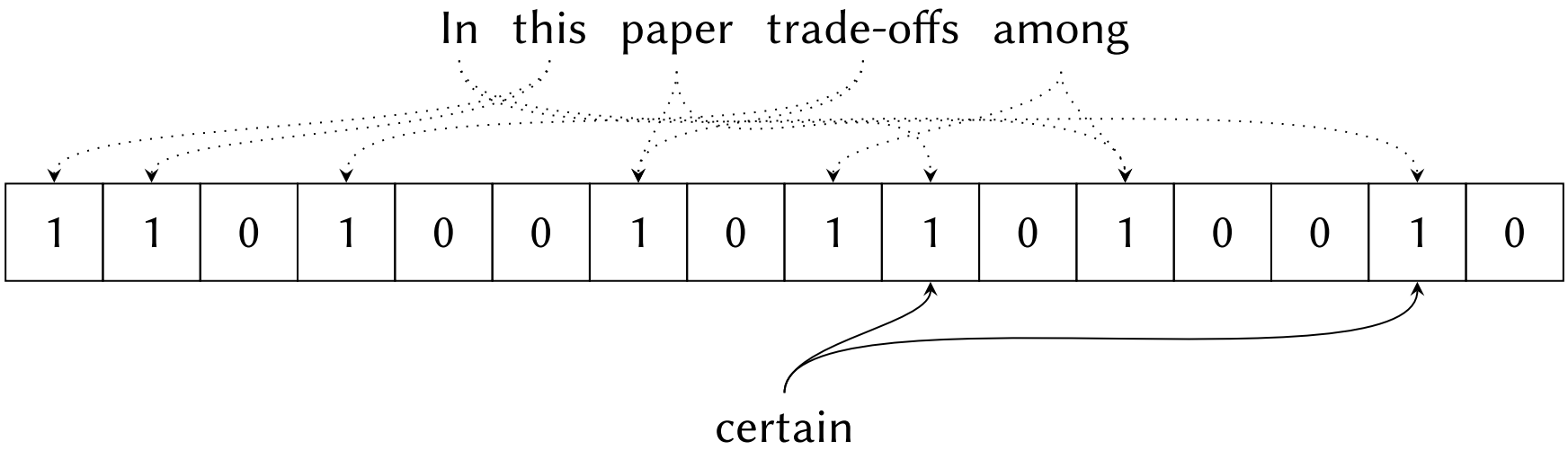
16 bit Bloom filter, inserting “trade-offs” (positions 6, 3).

Bloom Filter Example (6)



16 bit Bloom Filter, inserting “among” (positions 11, 8).

Bloom Filter Example (7)



16 bits Bloom filter, inserting “certain” (position 9, 14): false positive.

Searching in the Filter

- For each of “In”, “this”, “paper”, “trade-offs”, “among”, the filter will report, correctly, that they belong to the set.
- For “certain” the filter will report, falsely, that it belongs to the set.
- If for an element the filter reports that it does not belong to the set, it is certain that it does not belong to the set.

The Current Situation

- We have a false positive after five insertions, so we have a $1/6 \approx 17\%$ probability of false positives.
- We achieved this with 16 bits.
- If we were using a normal hash table, we would need the space for the hash table *and* the table for the keys that we would store in it.
- The keys require 33 bytes = 264 bits.
- That means that we achieved savings in the order of $264/16 = 16.5$ times!

Bloom Filter Parameters

A Bloom filter is characterized by the following parameters:

- m, the size of the filter (number of bits).
- n, the number of elements we expect.
- k, the number of hash functions we use.

Bloom Filter Analysis (1)

- Suppose that we have one hash function.
- The probability that a particular bit will remain zero is:

$$1 - \frac{1}{m}$$

- If we have k hash functions, the probability that a particular bit will remain zero is:

$$\left(1 - \frac{1}{m}\right)^k$$

Bloom Filter Analysis (2)

- We can rewrite the last formula as:

$$\left(1 - \frac{1}{m}\right)^{m\left(\frac{k}{m}\right)}$$

- When $m \longrightarrow \infty$ we have:

$$\left(1 - \frac{1}{m}\right)^{m} = 1/e = e^{-1}$$

- Therefore, the probability that a particular bit will be zero is:

$$e^{-k/m}$$

Bloom Filter Analysis (3)

- After inserting n elements, the probability that a particular bit will be zero is:

$$e^{-nk/m}$$

- The probability that a particular bit will be one is:

$$\left(1 - e^{-kn/m}\right)$$

- The probability that k particular bits will be one is:

$$\left(1 - e^{-kn/m}\right)^k$$

- That is the probability of a false positive.

Optimal Parameter Values

- From the last expression we can calculate the optimal values of the filter.
- If we have a given m (filter size) and n (number of elements), the best value for the number of hash functions is:

$$k = \frac{m}{n} \ln 2$$

- If we have a given number of elements n and a target false positive value of p, the required filter size is:

$$m = -\frac{n \ln p}{(\ln 2)^2}$$

Example of k

- If we want to handle 1,000,000,000 elements using a table of 10,000,000,000 bits, or 1.25 GBytes, we get:

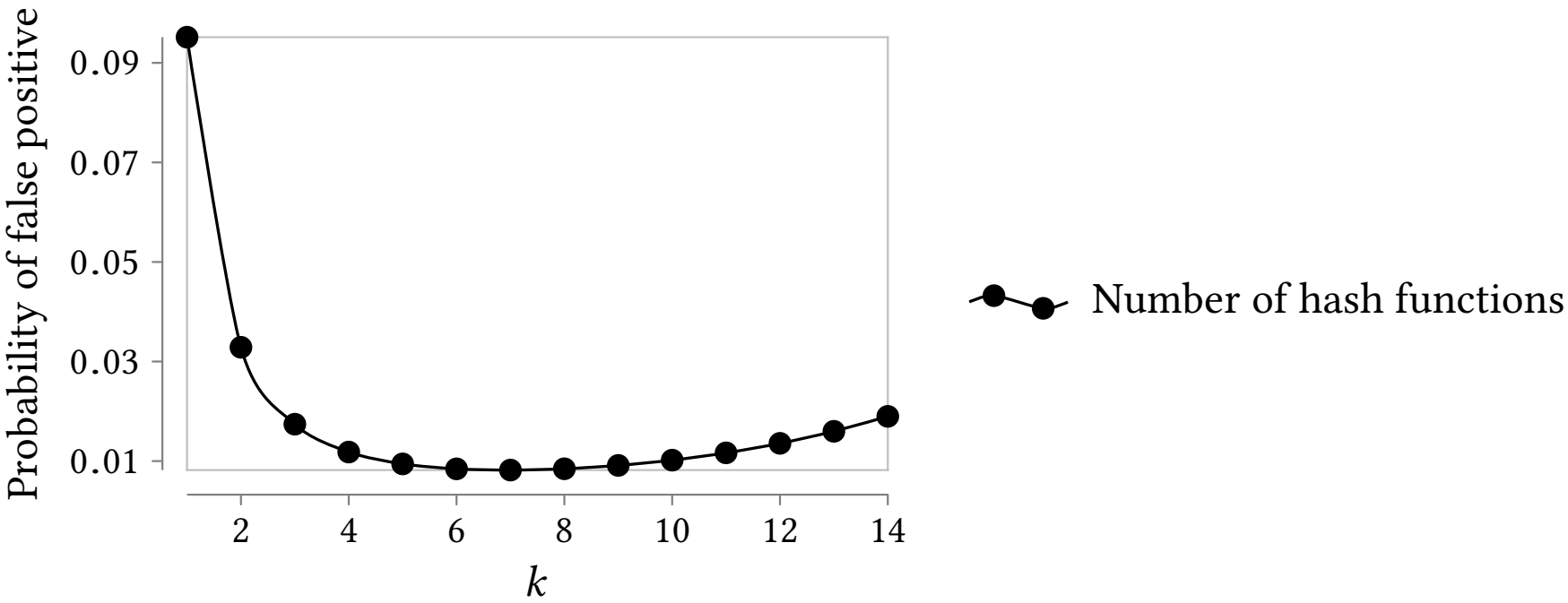
$$k = \frac{10^{10}}{10^9} \ln 2 \approx 7$$

- So we need seven hash functions.
- The false positive rate will be:

$$\left(1 - e^{-7\frac{10^9}{10^{10}}}\right)^7 \approx 0.008$$

or 8‰.

Behavior of k



We see that although seven functions is the optimal value, there is not much difference from five functions.

Example of m

- If we want to handle 1,000,000,000 elements with a false positive rate of 1%, we get:

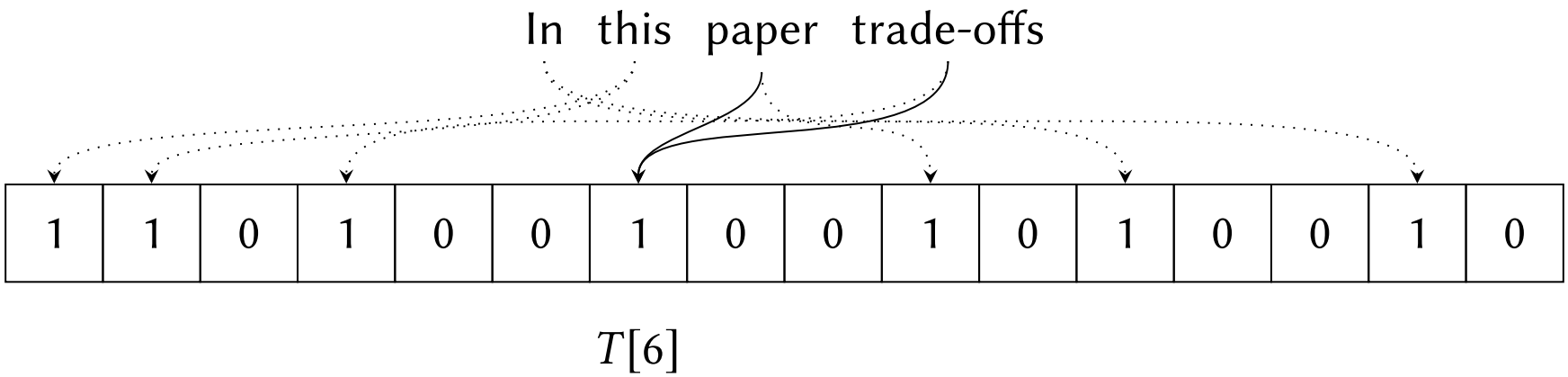
$$m = -\frac{n \ln p}{(\ln 2)^2} \approx 9.585.058.378$$

- That is, we need a table of about 10 billion bits, or 1,250,000,000 bytes, or 1.25 Gbytes.
- So, with 1.25 GBytes we can handle 1 billion elements.
- If we consider that each element can consist of 10 bytes, then with 1.25 GBytes we can handle 10 Gbytes with a computational cost equal to just the calculation of some hash functions!

Counting Bloom Filters

- In simple Bloom filters we cannot delete an element, because we do not know how many different elements hash to the same position in the table.
- This happens to any "partial" collision, where two or more elements hash to at least one position in common.
- If x_1, x_2 are two elements, this happens when $h_i(x_1) = h_j(x_2)$ for some i and j.

The Deletion Problem



If we delete “paper” or “trade-offs”, we’ll set $T[6]$ equal to zero; but this also corresponds to the other item.

The Solution: Counting Bloom Filters

- Instead of using a bit array, we use an array where in each position we store a counter.
- Initially the array has zero in all positions.
- When we insert elements, we increase the corresponding counters by one.
- When we remove elements, we reduce the corresponding counters by one.
- Membership check is as before.

Counting Bloom Filter Insertion

Algorithm: Insert into counting Bloom filter.

`InsertInCntBloomFilter`(T, x)

Input: T, an integer array of size m

x, a record to insert to the set represented by the counting Bloom filter

Result: the record is inserted in the counting Bloom filter by increasing by 1 the counters at $h_0(x), h_1(x), \ldots, h_{k-1}(x)$

```
1  for i ← 0 to k do
2      h ← h_i(x)
3      T[h] ← T[h] + 1
```

Deletion from Counting Bloom Filter

Algorithm: Remove from a counting Bloom filter.

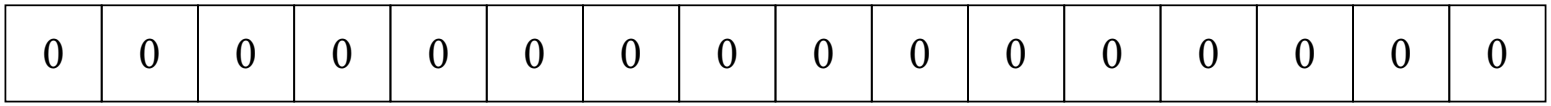
`RemoveFromCntBloomFilter`(T, x)

Input: T, an integer array of size m
x, a record to remove from the set represented by the counting Bloom filter

Result: the record is removed from the counting Bloom filter by decreasing by 1 the counters at $h_0(x), h_1(x), \ldots, h_{k-1}(x)$

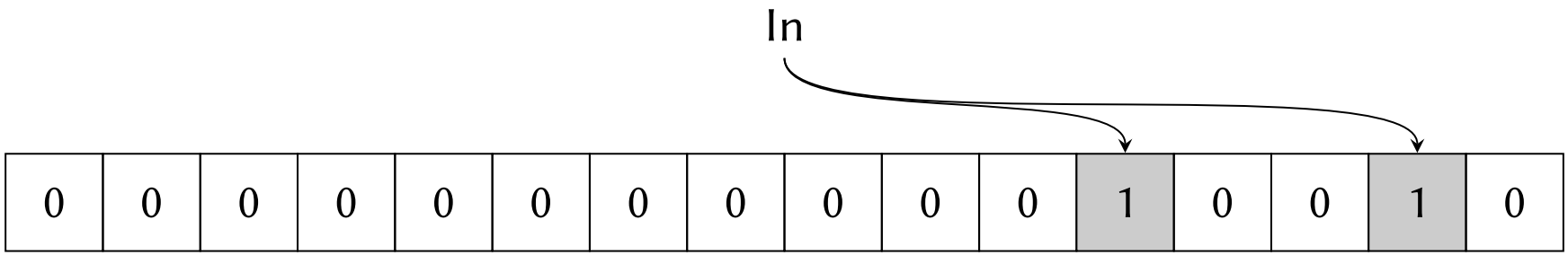
```
1  for i ← 0 to k do
2      h ← h_i(x)
3      if T[h] ≠ 0 then
4          T[h] ← T[h] − 1
```

Counting Bloom Filter Example (1)



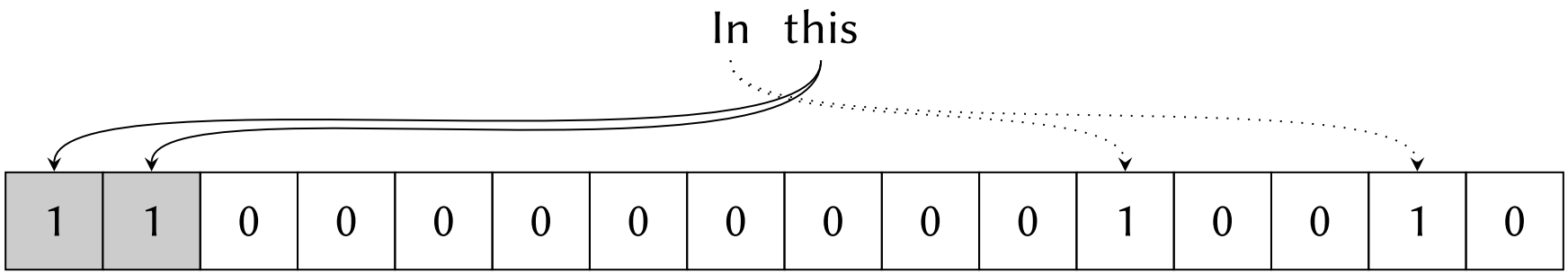
16 bit counting Bloom filter, initial state.

Counting Bloom Filter Example (2)



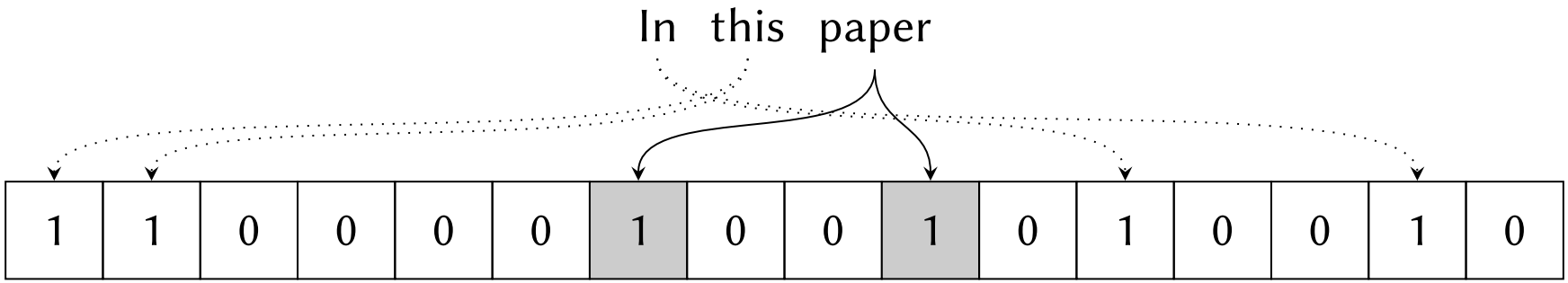
16 bits Bloom filter, inserting “In”.

Counting Bloom Filter Example (3)



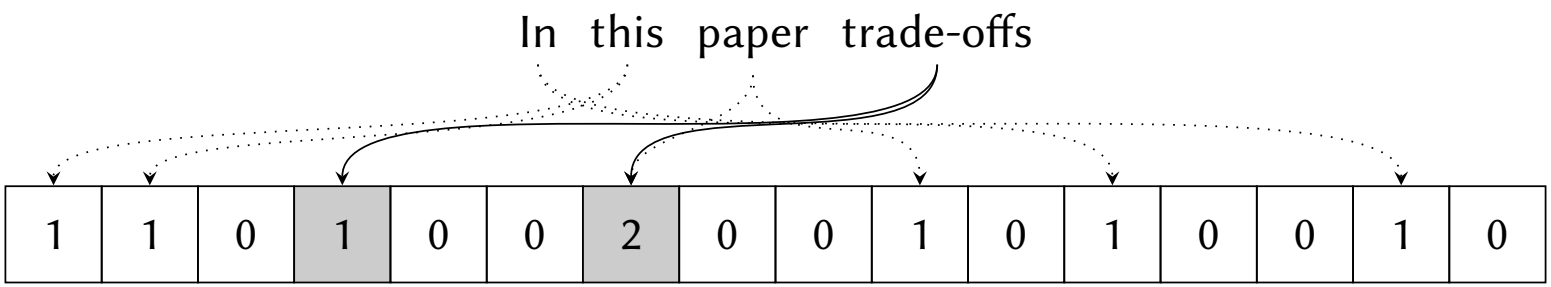
16 bits counting Filter, inserting “this”.

Counting Bloom Filter (4)



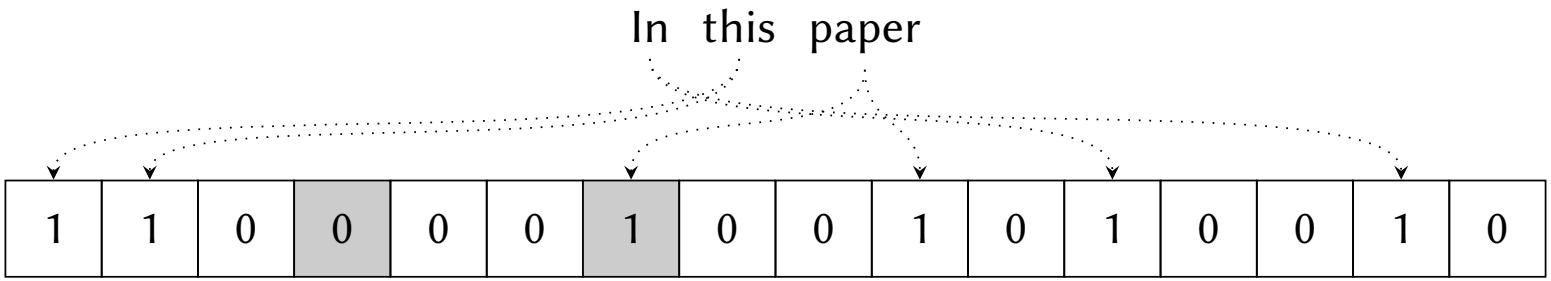
16 bit counting Bloom filter, inserting “paper” (positions 9, 6).

Counting Bloom Filter Example (5)



16 bit counting Bloom filter, inserting “trade-offs”.

Counting Bloom Filter Example (5)



16 bit counting Bloom filter, deletion of “trade-offs”.

Hash Functions for Bloom Filters

- To implement Bloom filters we need hash functions that can return more than one hash value for the same item.
- These functions will take as input the item that we want to hash and an additional parameter.
- There are several such hash functions: MurmurHash, Jenkins, FarmHash, ...

FNV-1A-based Hash

Algorithm: FNV-1a based 32 bits hash.

FNV-1a$(s, i) \longrightarrow h$

Input: s, a string
i, an integer

Output: the 32 bits hash value of s

1 $h \leftarrow$ 0x811C9DC5
2 $p \leftarrow$ 0x01000193
3 $h \leftarrow h \oplus i$
4 **foreach** c **in** s **do**
5 $h \leftarrow h \oplus$ Ordinal(c)
6 $h \leftarrow (h \times p)$ & 0xFFFFFFFF
7 **return** h

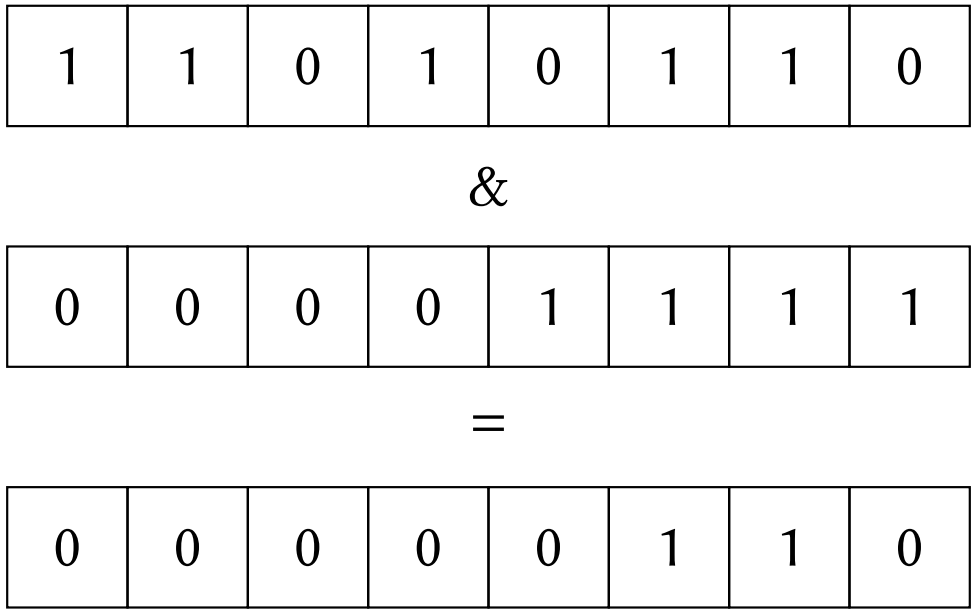
This algorithm is based on the FNV-1a algorithm, also known as Fowler/Noll/Vo from the names of its inventors, Glenn Fowler, Landon Curt Noll, and Phong Vo.

Bitwise AND

		x	
		0	1
y	0	0	0
	1	0	1

The bitwise AND operation.

Applying a Bit Mask with AND

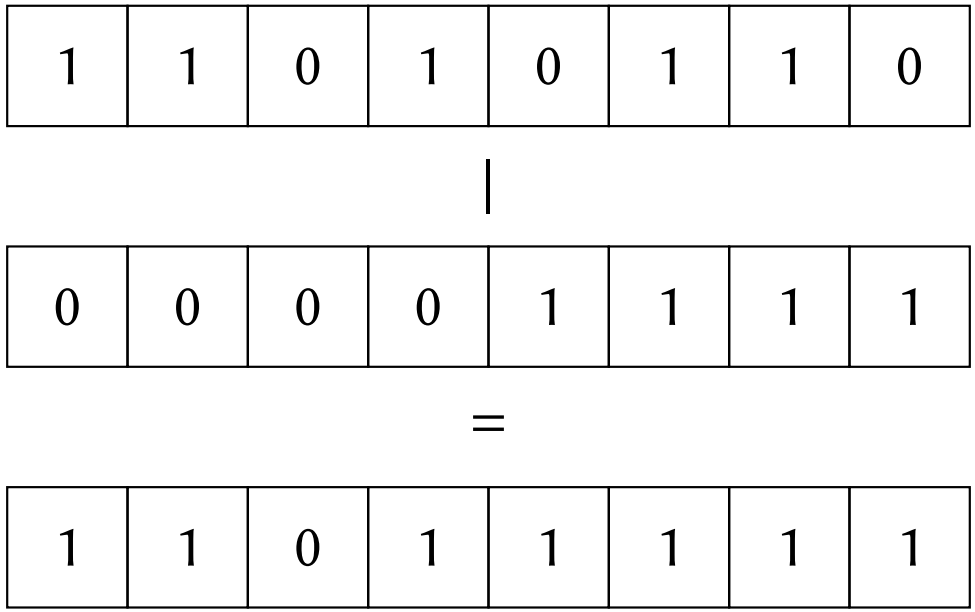


Applying bit mask 00001111 = 0xF to 11010110 with bitwise AND.

Bitwise OR

		x	
		0	1
y	0	0	1
	1	1	1

Applying a Bitwise Mask with OR



Applying bit mask 00001111 = 0xF to 11010110 with bitwise OR.