Hash Table

Introduction to Algorithms

Anakkapon Saenthon

Overview

1. Hashing
2. Collision problem
3. Collision Hashing Techniques
4. Demonstrate with Python

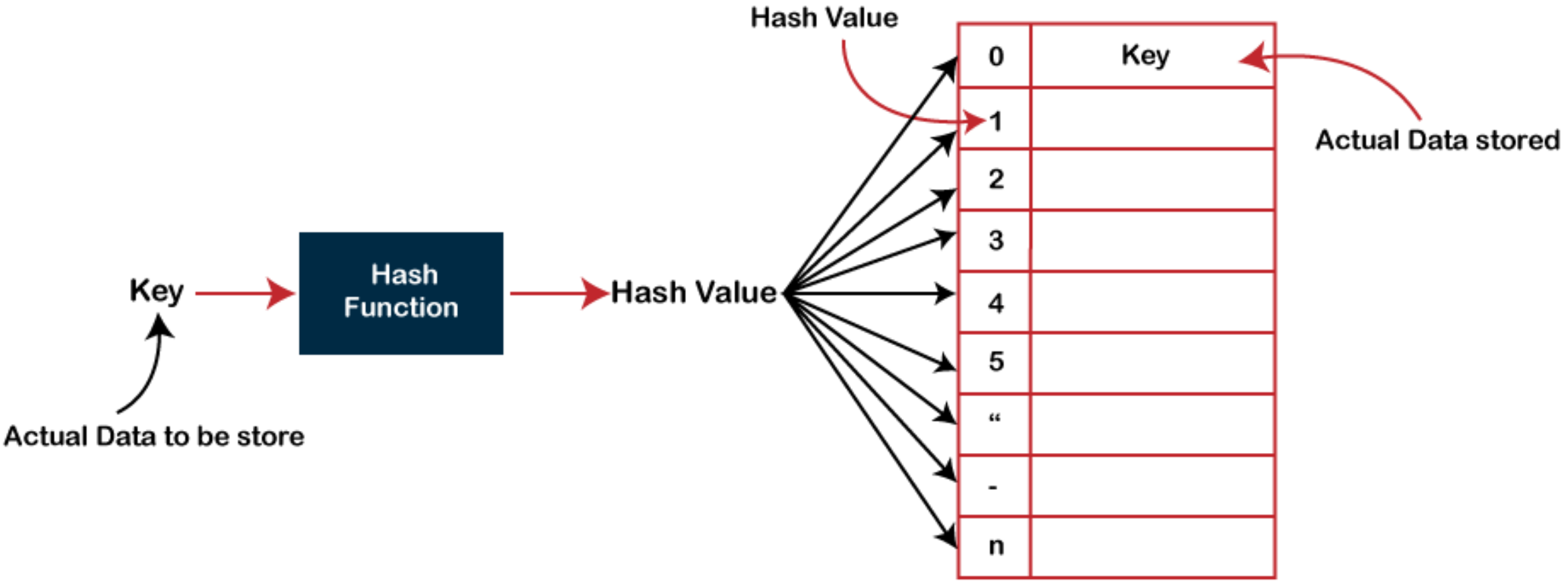
What is Hashing?

- Hashing is a data structure technique or process of **mapping keys**, **values** into the hash table by using a hash function.

- It is done for **faster access to elements**.

- The efficiency of mapping depends on the efficiency of the hash function used.

What is Hashing?

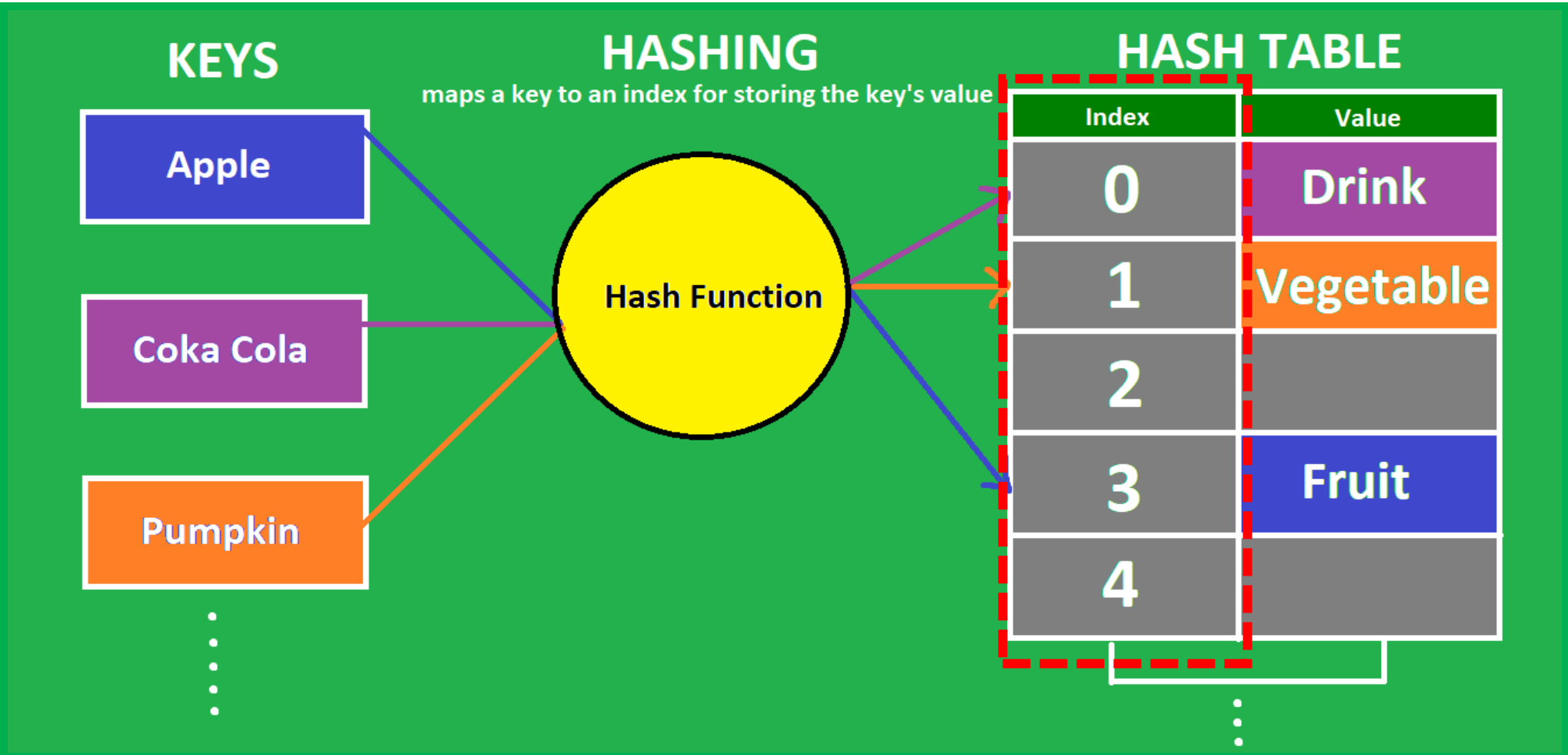


Key : integer, string, …

How we use Hash table ?

Use a key (arbitrary string or number) to index directly into an array

– O(1) time to access records



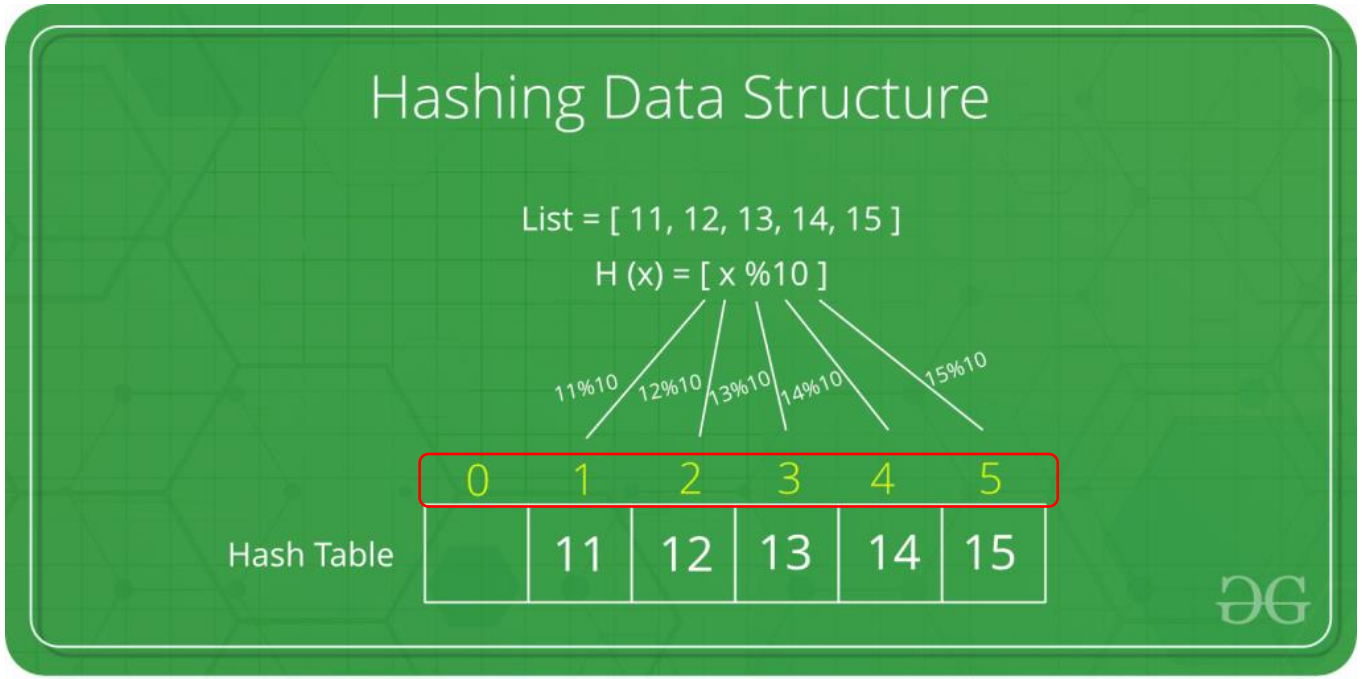
How we fine the hash key (index) ?

Example: **integer as key**

Calculate with *Hash Function*

Hash Function

H(x) = X % 10



Input : **Integer**

[11 12 13 14 15]

→

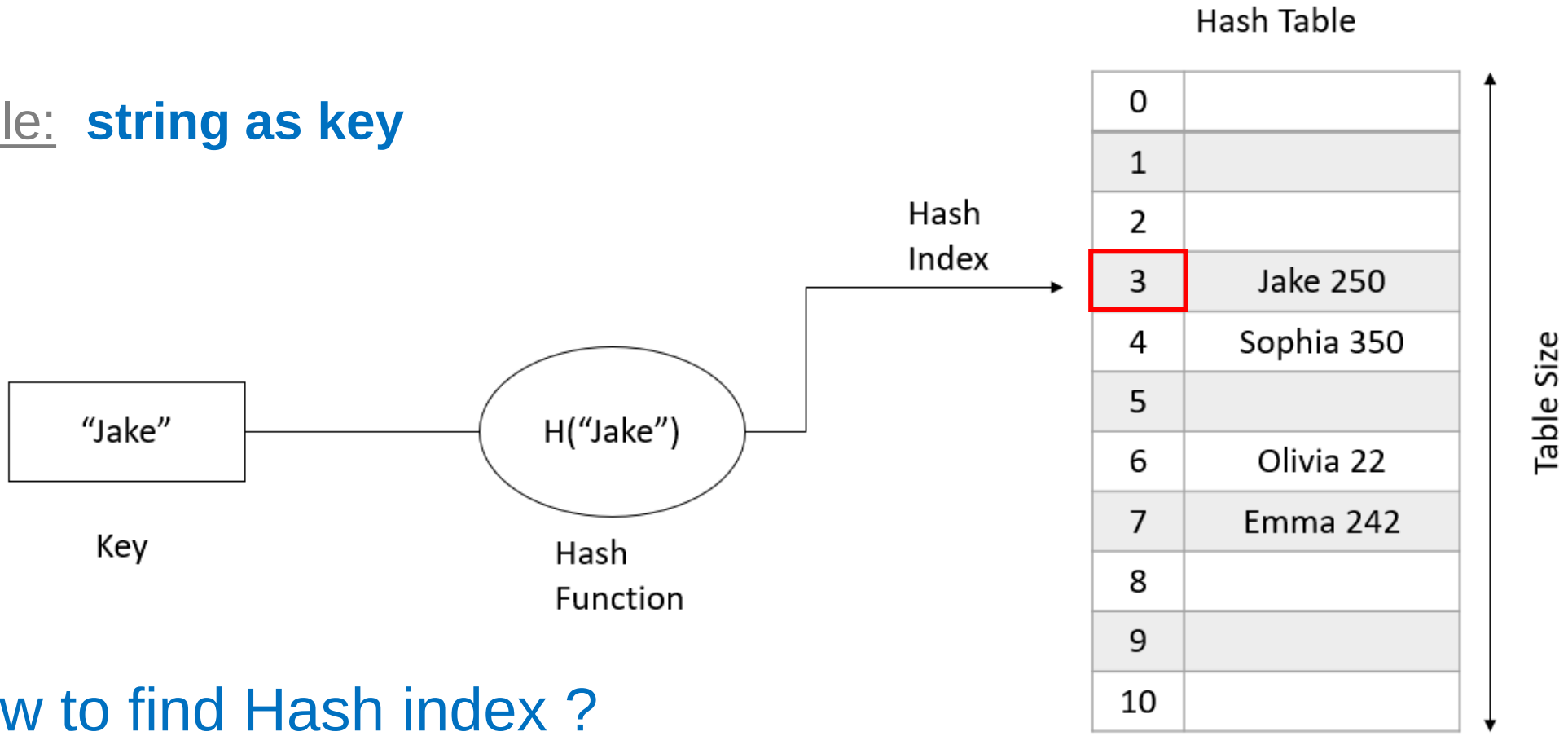
hash key

[1 2 3 4 5]

Hasing

We generate a hash for the input using the hash function and then store the element using the generated hash as the key in the hash table.

Example: **string as key**



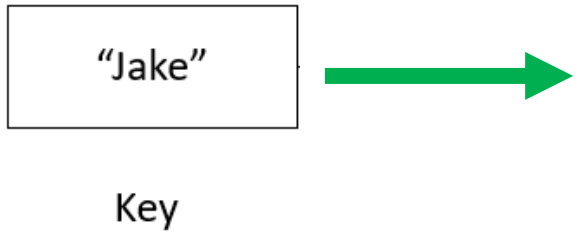
How to find Hash index ?

ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	`
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(	72	48	110	H	104	68	150	h
9	9	11		41	29	51	)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[	123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	|
29	1D	35		61	3D	75	=	93	5D	135	]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	_	127	7F	177	

Example : Find the Hash index

ASCII values of characters in *key*



String	Look ASCII Table
J	074
a	097
k	107
e	101

= 379

Dec	Hex	Oct	Char
64	40	100	@
65	41	101	A
66	42	102	B
67	43	103	C
68	44	104	D
69	45	105	E
70	46	106	F
71	47	107	G
72	48	110	H
73	49	111	I
74	4A	112	J
75	4B	113	K

Dec	Hex	Oct	Char
96	60	140	`
97	61	141	a
98	62	142	b
99	63	143	c
100	64	144	d
101	65	145	e
102	66	146	f
103	67	147	g
104	68	150	h
105	69	151	i
106	6A	152	j
107	6B	153	k

What is the table size ?

What the value of this string ?

John — 74+111+104+156 = ???

Peter — 80+101+116+101+114 = ???

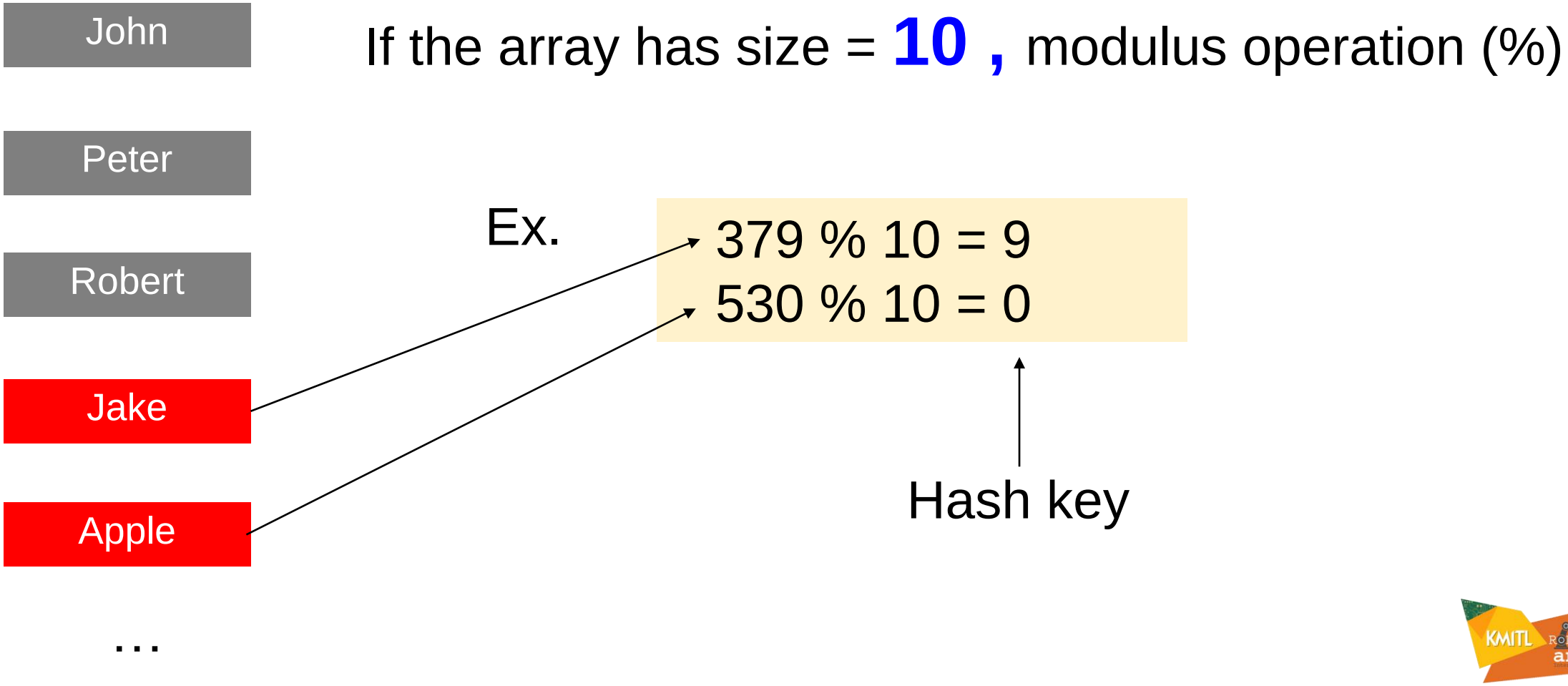
Robert — ???

Jake — = 379

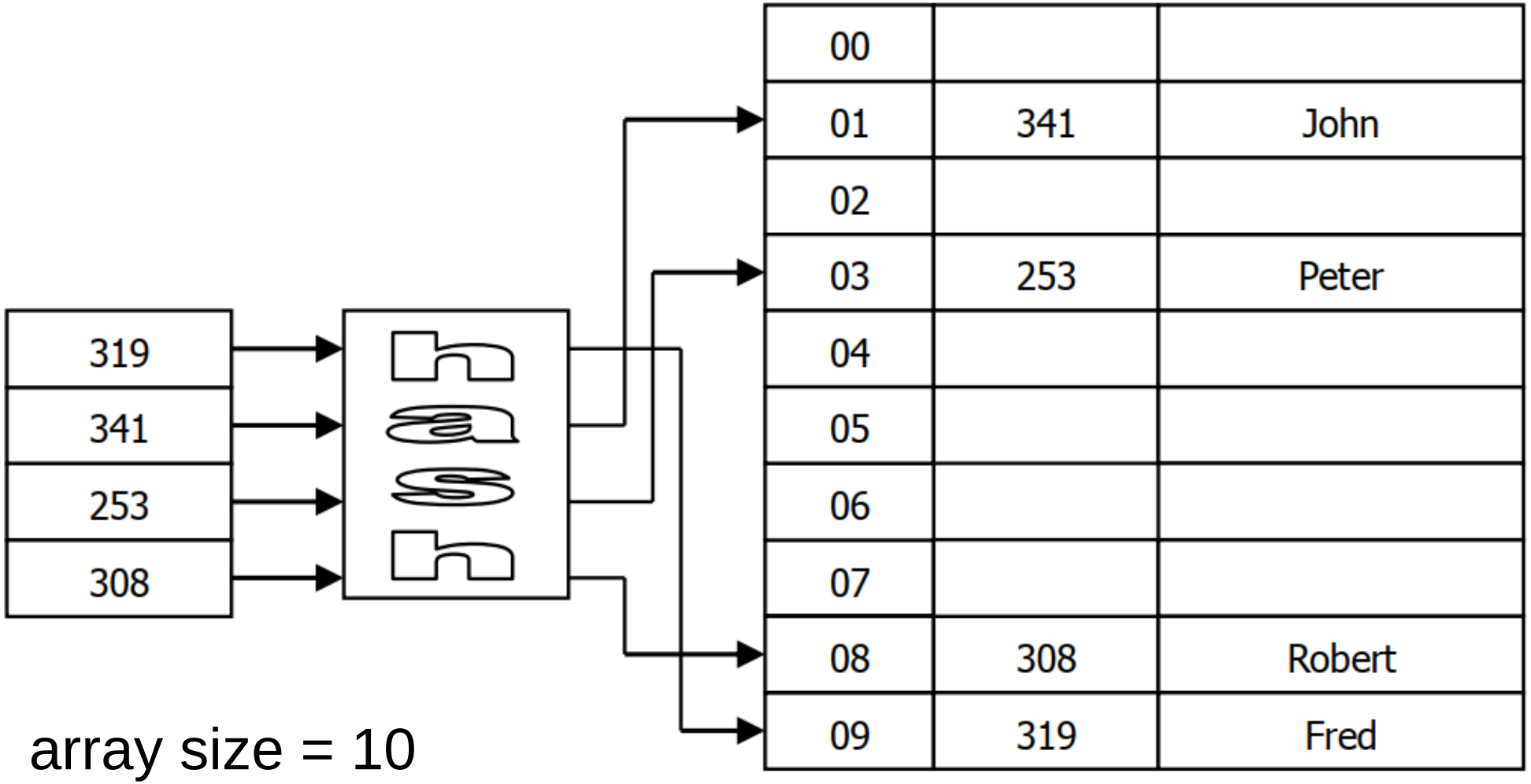
Apple — = 530

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
64	40	100	@	96	60	140	`
65	41	101	A	97	61	141	a
66	42	102	B	98	62	142	b
67	43	103	C	99	63	143	c
68	44	104	D	100	64	144	d
69	45	105	E	101	65	145	e
70	46	106	F	102	66	146	f
71	47	107	G	103	67	147	g
72	48	110	H	104	68	150	h
73	49	111	I	105	69	151	i
74	4A	112	J	106	6A	152	j
75	4B	113	K	107	6B	153	k
76	4C	114	L	108	6C	154	l
77	4D	115	M	109	6D	155	m
78	4E	116	N	110	6E	156	n
79	4F	117	O	111	6F	157	o
80	50	120	P	112	70	160	p
81	51	121	Q	113	71	161	q
82	52	122	R	114	72	162	r
83	53	123	S	115	73	163	s
84	54	124	T	116	74	164	t
85	55	125	U	117	75	165	u
86	56	126	V	118	76	166	v
87	57	127	W	119	77	167	w
88	58	130	X	120	78	170	x
89	59	131	Y	121	79	171	y
90	5A	132	Z	122	7A	172	z
91	5B	133	[	123	7B	173	{
92	5C	134	\	124	7C	174	|
93	5D	135	]	125	7D	175	}
94	5E	136	^	126	7E	176	~
95	5F	137	_	127	7F	177	

How to reduce the table size ?



Example of the table size (input string)



array size = 10

Collision

hash table

What is Collision ?

- When two keys have the same hash value.
  - 312 % 10 = 2
  - 322 % 10 = 2 !!!

- Because the hash method is not guaranteed to return a unique integer for each key

Collision Hashing Techniques

There are several ways to handle collisions :

1. Linear Probing
2. Quadratic Probing
3. Double Hashing
4. Linked-list

1. Linear Probing

- ***Linear Probing***: search sequentially for an unoccupied position
  - uses a wraparound (circular) array

Algorithm : forward by 1, until not occur collision

- Probe sequence:

  0^{th} probe = h(k) mod TableSize

  1^{th} probe = (h(k) + **1**) mod TableSize

  2^{th} probe = (h(k) + **2**) mod TableSize

  . . .

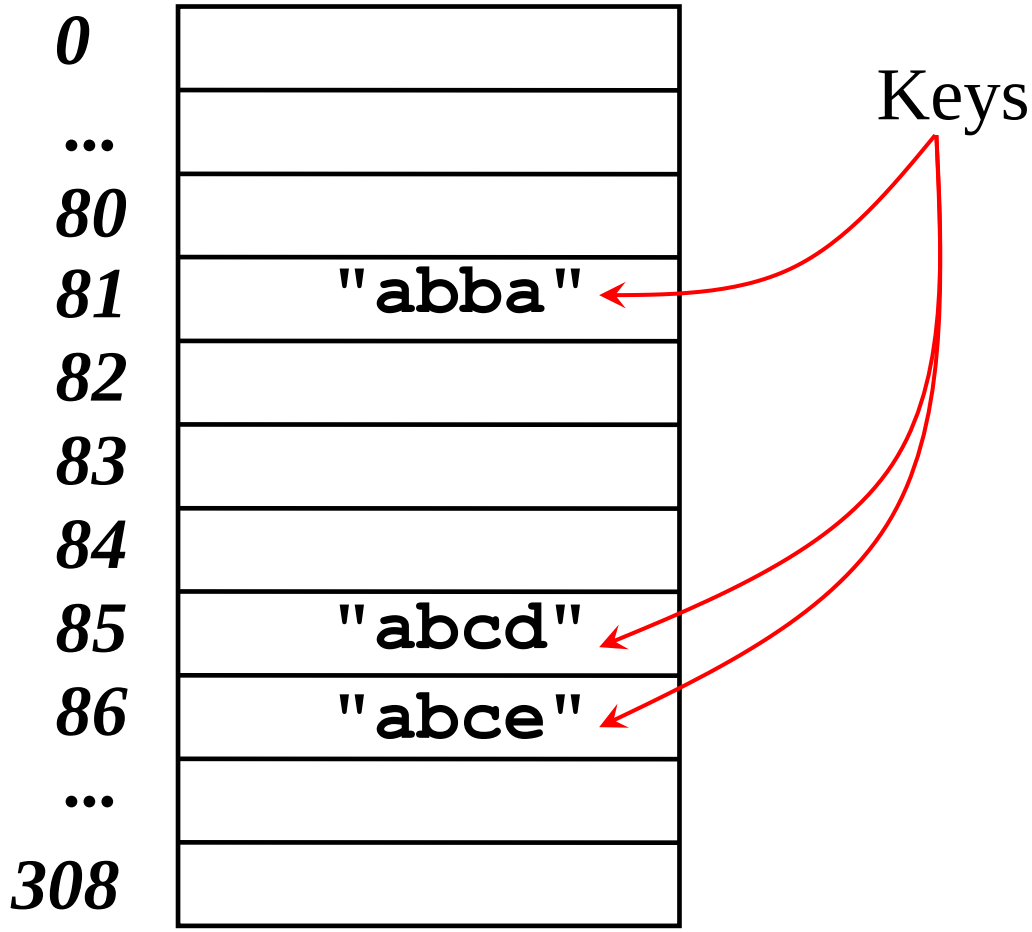
  i^{th} probe = (h(k) + **i**) mod TableSize

A hash table after three insertions

using the too simple (lousy) hash method

insert objects
with these
three keys:

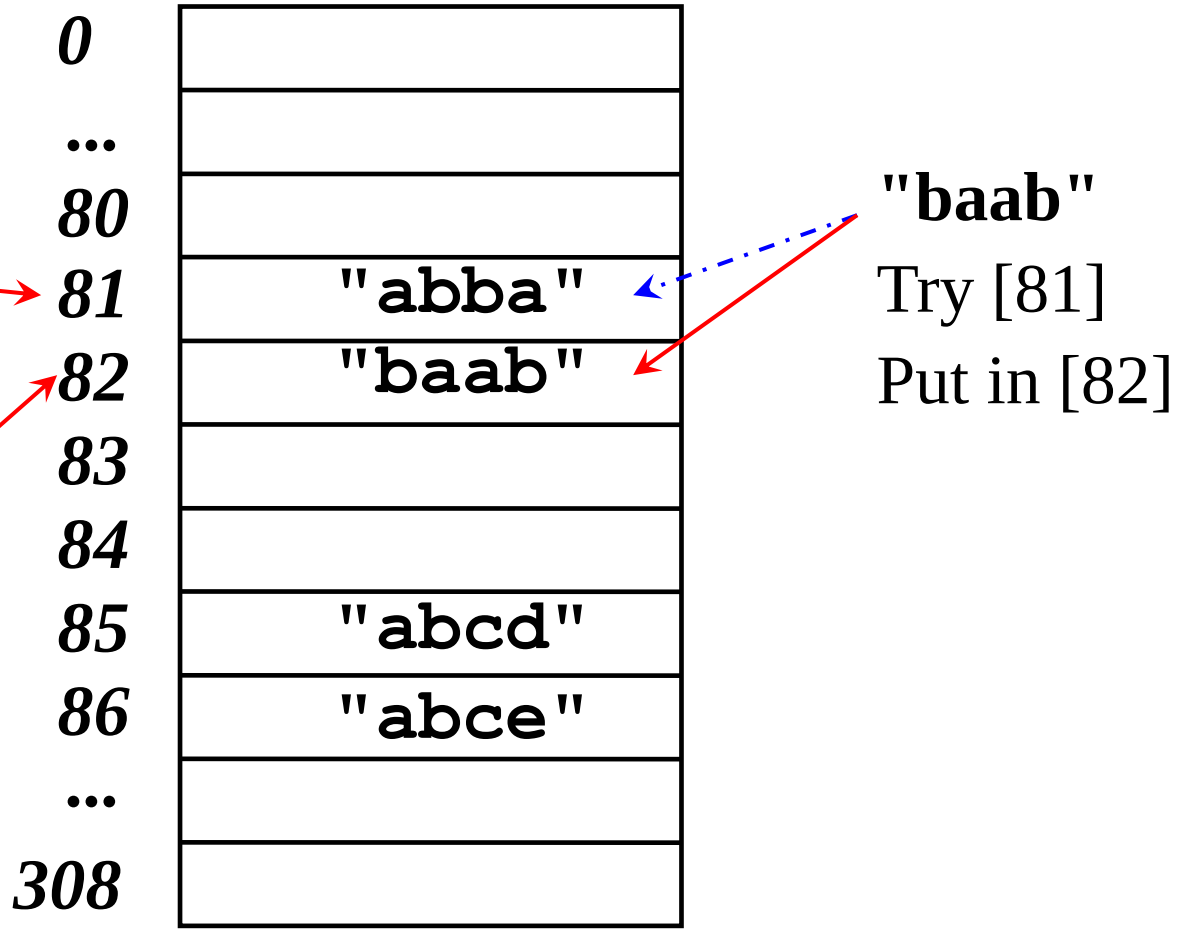
"abba"
"abcd"
"abce"



Collision occurs while inserting "baab"

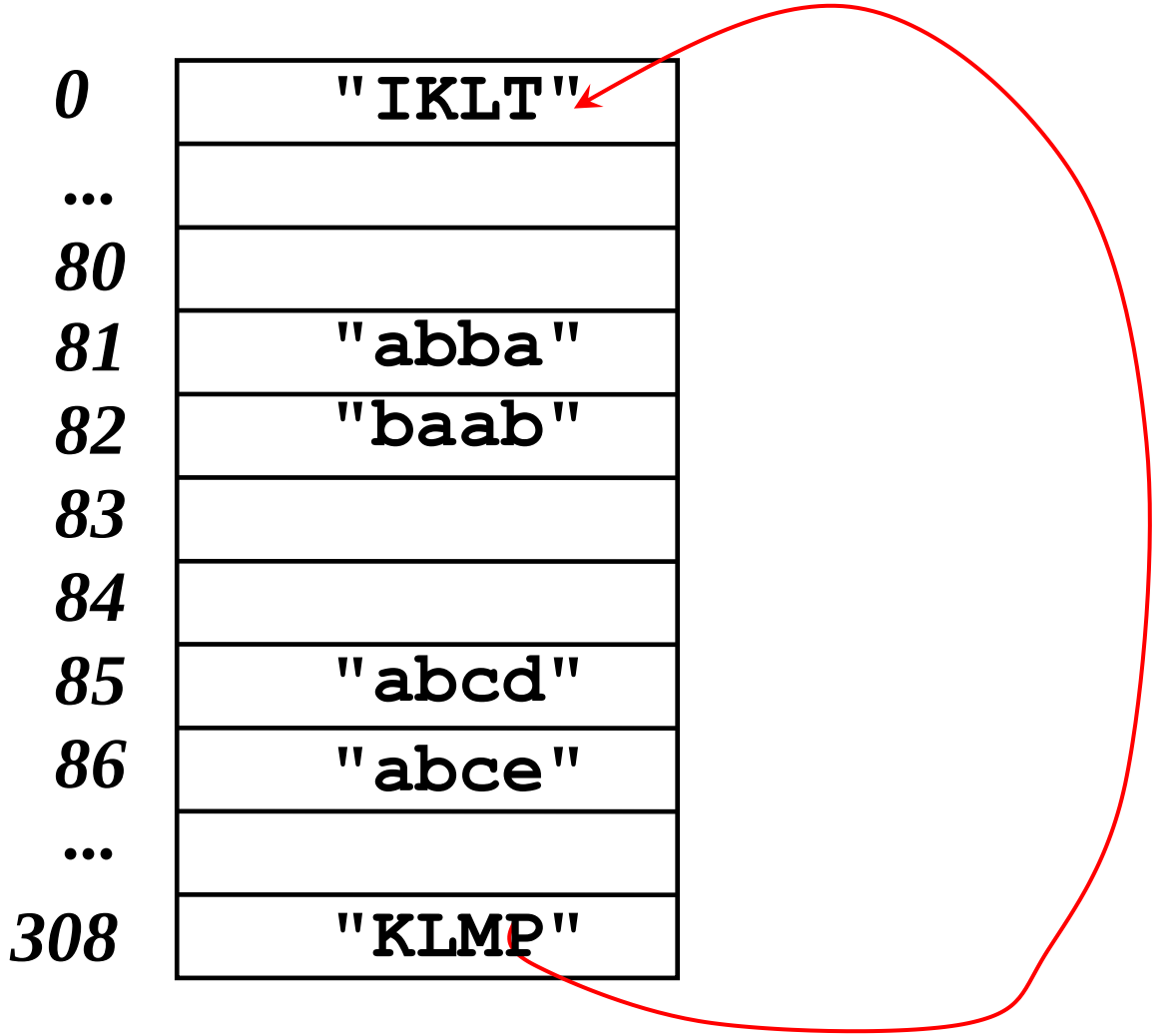
can't insert "baab" where it hashes to same slot as "abba"

Linear probe forward by 1, inserting it at the next available slot



Wrap around when collision occurs at end

Insert "**KLMP**" and "**IKLT**" both of which have a hash value of 308

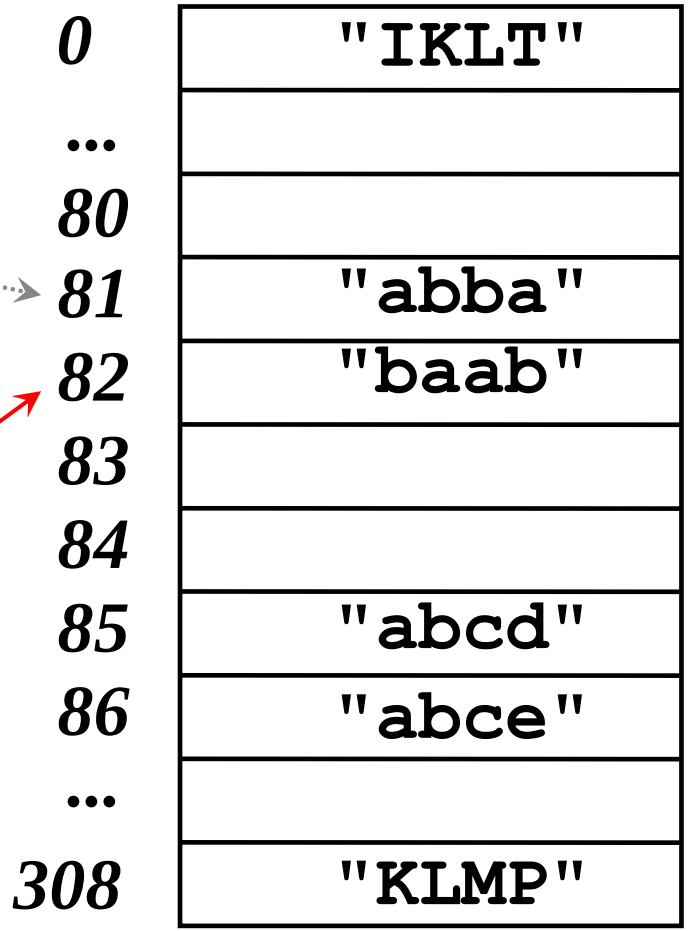


Wrap around (circular)

How to Find object with key "baab"

"baab" still hashes to 81, but since [81] is occupied, linear probe to [82]

At this point, you could return a reference or remove baab



2. Quadratic Probing

Quadratic probing eliminates the primary clustering problem

Algorithm : add index values in increments of powers of 2

- Probe sequence:

  0^{th} probe = h(k) mod TableSize

  1^{th} probe = (h(k) + **1**) mod TableSize

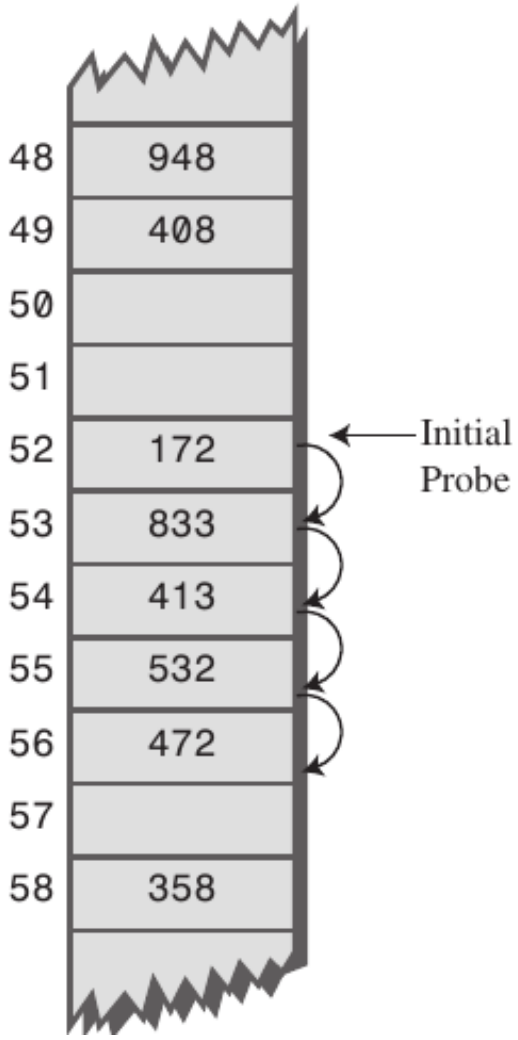
  2^{th} probe = (h(k) + **4**) mod TableSize

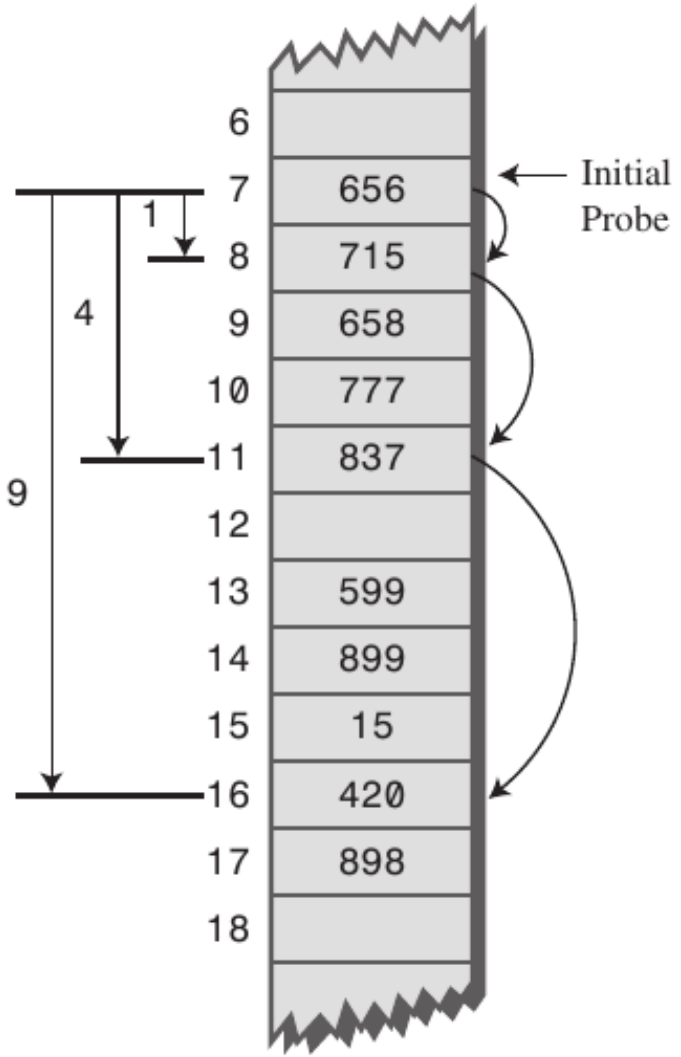
  3^{th} probe = (h(k) + **9**) mod TableSize

  . . .

  i^{th} probe = (h(k) + $\mathbf{i^2}$) mod TableSize

Linear vs Quadratic probing





3. Double Hashing

$$f(i) = i * g(k)$$

where g is a second hash function

Probe sequence:

0^{th} probe = h(k) mod TableSize

1^{th} probe = (h(k) + **g(k)**) mod TableSize

2^{th} probe = (h(k) + **2*g(k)**) mod TableSize

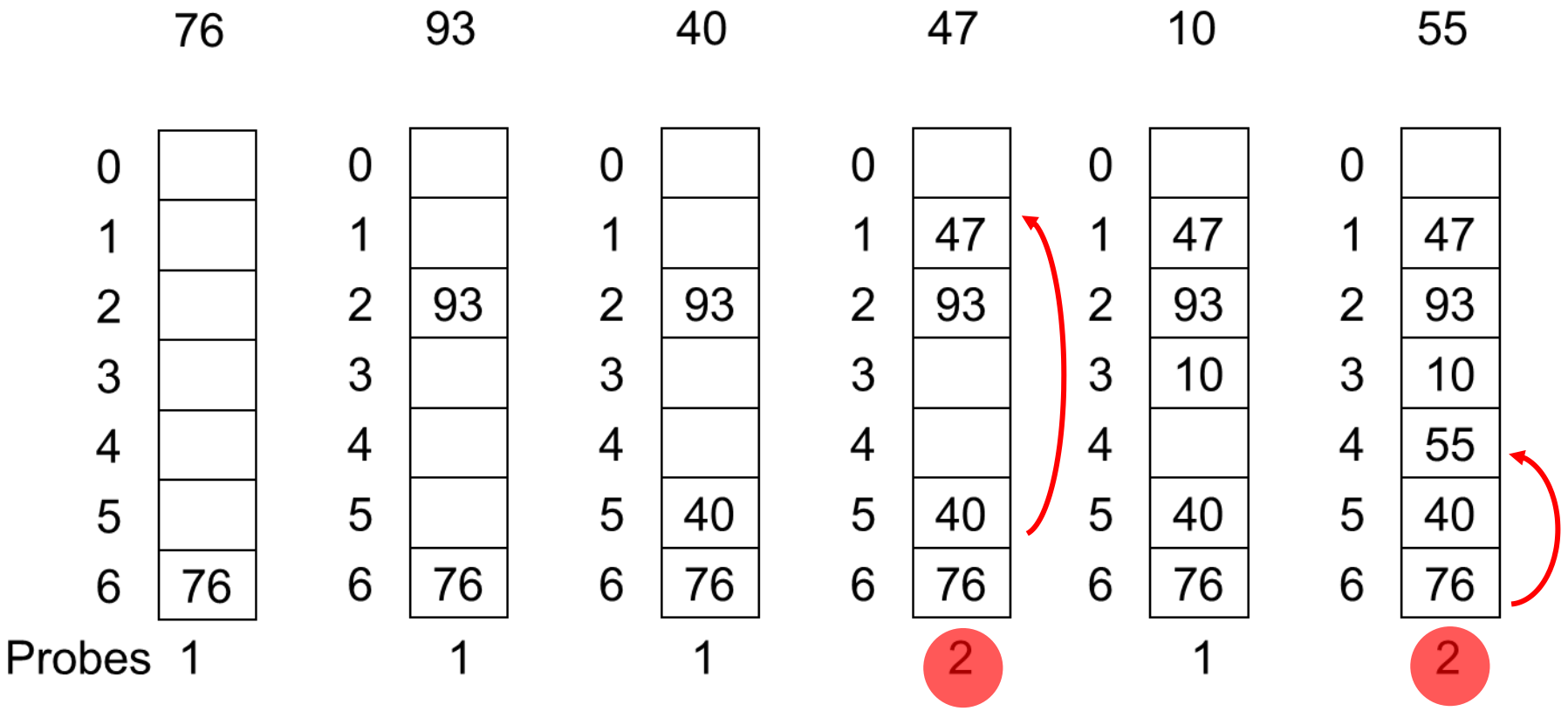
3^{th} probe = (h(k) + **3*g(k)**) mod TableSize

. . .

i^{th} probe = (h(k) + **i*g(k)**) mod TableSize

Double Hashing Example

h(k) = k mod 7 , g(k) = 5 – (k mod 5)



Hash index = h(k) + **2*g(k)**)

5+(5-2)= 5+3 = 8

6+(5-0)= 6+5 = 11

4. Linked-list (Separate Chaining)

The idea is to make each cell of hash table point to a linked list of records that have same hash function value

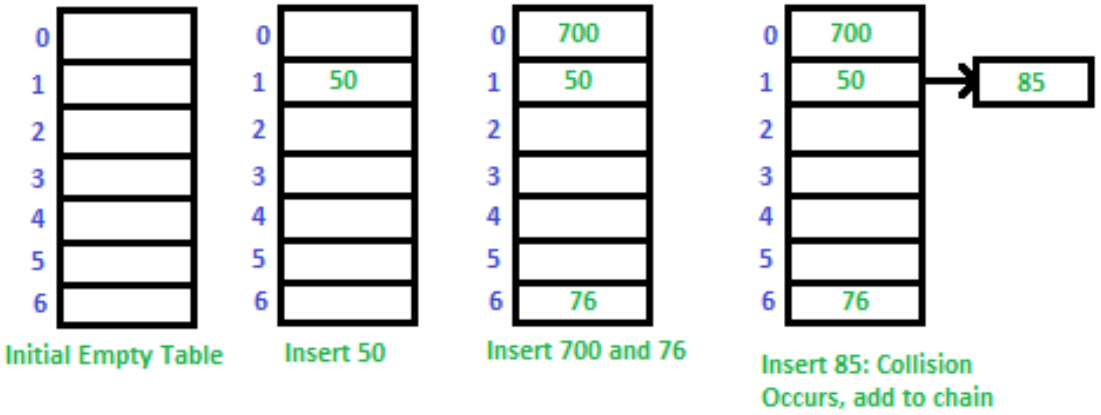
Advantages:

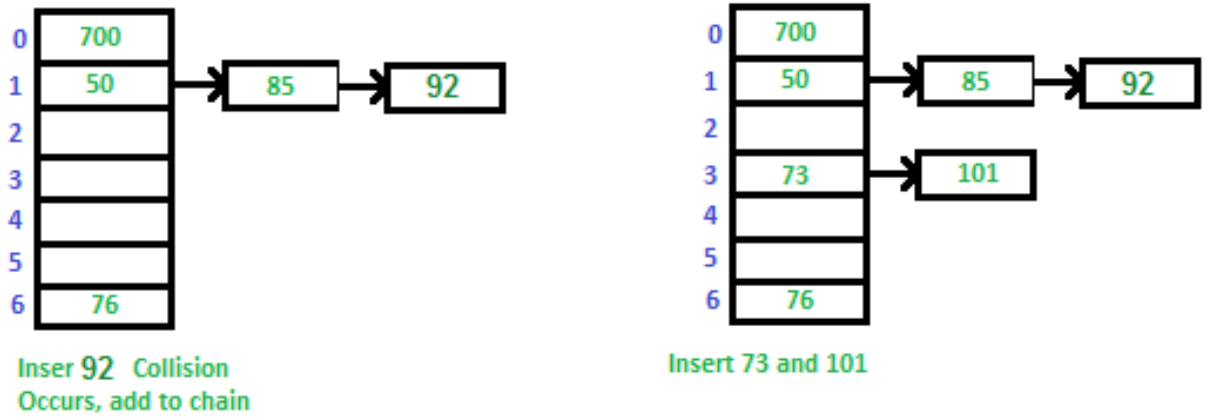
1) Simple to implement.
2) Hash table never fills up, we can always add more elements to the chain.
3) Less sensitive to the hash function or load factors.
4) It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

4. Linked-list (Separate Chaining)

Let us consider a simple hash function as "**key mod 7**" and sequence of keys as

50, 700, 76, 85, 92, 73, 101.





Real-world Applications

- Databases
- Associative arrays
- Sets
- Memory cache

Advantages of hash tables

- Here, are benefits of using hash tables:

  - Hash tables have high performance when looking up data, inserting, and deleting existing values.
  - The time complexity for hash tables is constant regardless of the number of items in the table.
  - They perform very well even when working with large datasets.

Disadvantages of hash tables

- Here, are cons of using hash tables:

  - You **cannot use a null value** as a key.
  - Collisions cannot be avoided when generating keys using. hash functions. Collisions occur when a key that is already in use is generated.
  - If the hashing function has many collisions, this can lead to performance decrease.

Demonstrate with Python

- In Python, the **Dictionary data types** represent the implementation of hash tables. The Keys in the dictionary satisfy the following requirements.
  - The keys of the dictionary are hashable i.e. the are generated by hashing function which **generates unique result** for each unique value supplied to the hash function.
  - The order of data elements in a dictionary is not fixed.
- So we see the implementation of hash table by using the dictionary data types as b

Accessing Values in Dictionary

To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value.

Example

```
# Declare a dictionary
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
# Accessing the dictionary with its key
print (dict['Name']: ", dict['Name'] )
print (dict['Age']: ", dict['Age'])
```

Output

```
dict['Name']:  Zara
dict['Age']:  7
```

Updating Dictionary

You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing

Example

```
# Declare a dictionary
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First’}
dict['Age'] = 8;
# update existing entry
dict['School'] = "DPS School";
# Add new entry
print ("dict['Age']: ", dict['Age’])
print ("dict['School']: ", dict['School’])
```

Output

```
dict['Age']:  8
dict['School']:  DPS School
```

Delete Dictionary Elements

You can either remove individual dictionary elements or clear the entire contents of a dictionary.

Example

```
dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}
del dict['Name'];
# remove entry with key 'Name'
dict.clear();
# remove all entries in dict
del dict ;
# delete entire dictionary
print ("dict['Age']: ", dict['Age'])
print ("dict['School']: ", dict['School'])
```

Problems Assignment 9

* Coding with **Colab**

1. Demonstrate declare a dictionary elements

Key	Value
'name'	'Jack'
'age'	18
‘class’	Algorithms

2. Demonstrate update a dictionary and insert the elements

Key	Value
'name'	'RAI'
'age'	20
'year'	2021
‘class’	Introduction

3. Demonstrate delete a dictionary elements

Key	Value
'age'	20
‘class’	Introduction

Submitted before 11.59 (Sunday)

Q & A