Advanced Programming
INFO135

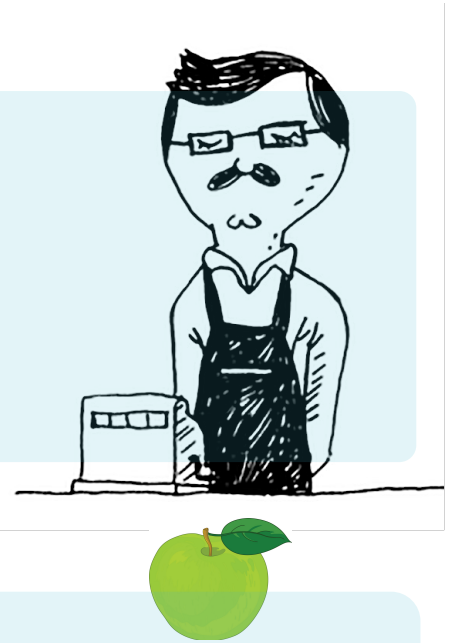
Lecture 6: Hashing

Mehdi Elahi

University of Bergen (UiB)

Hashing

- Suppose you work at a **grocery** store.
- When a customer buys a product (e.g. apple), you **look up price** in pricebook.

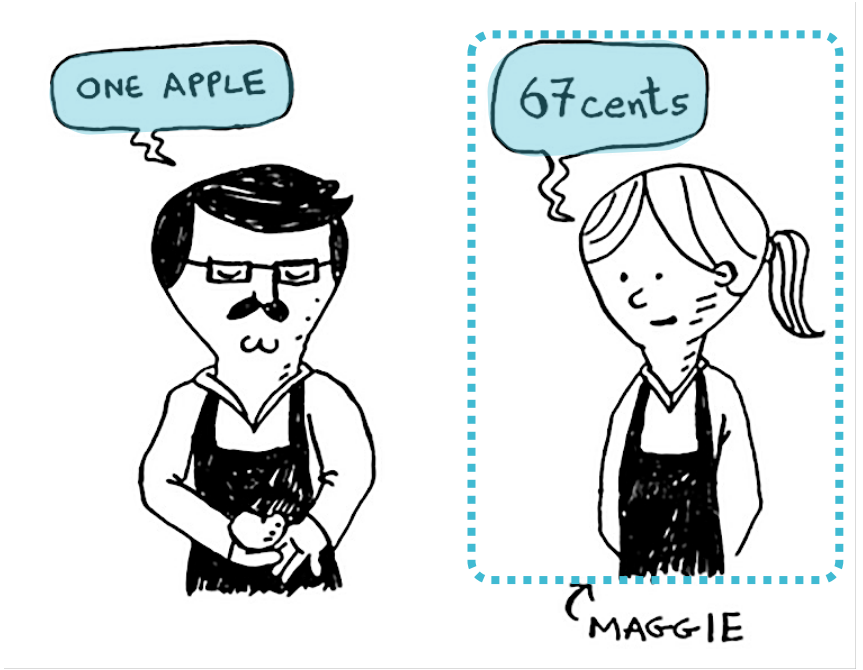


a) What if pricebook is **un-alphabetized**:
   - it would take a long time to look through every single line for apple: **O(n) time**.

b) What if the book is **alphabetized**:
   - it would be quick since you could run **Binary Search** to find the price: **O(log n) time**.

Hashing

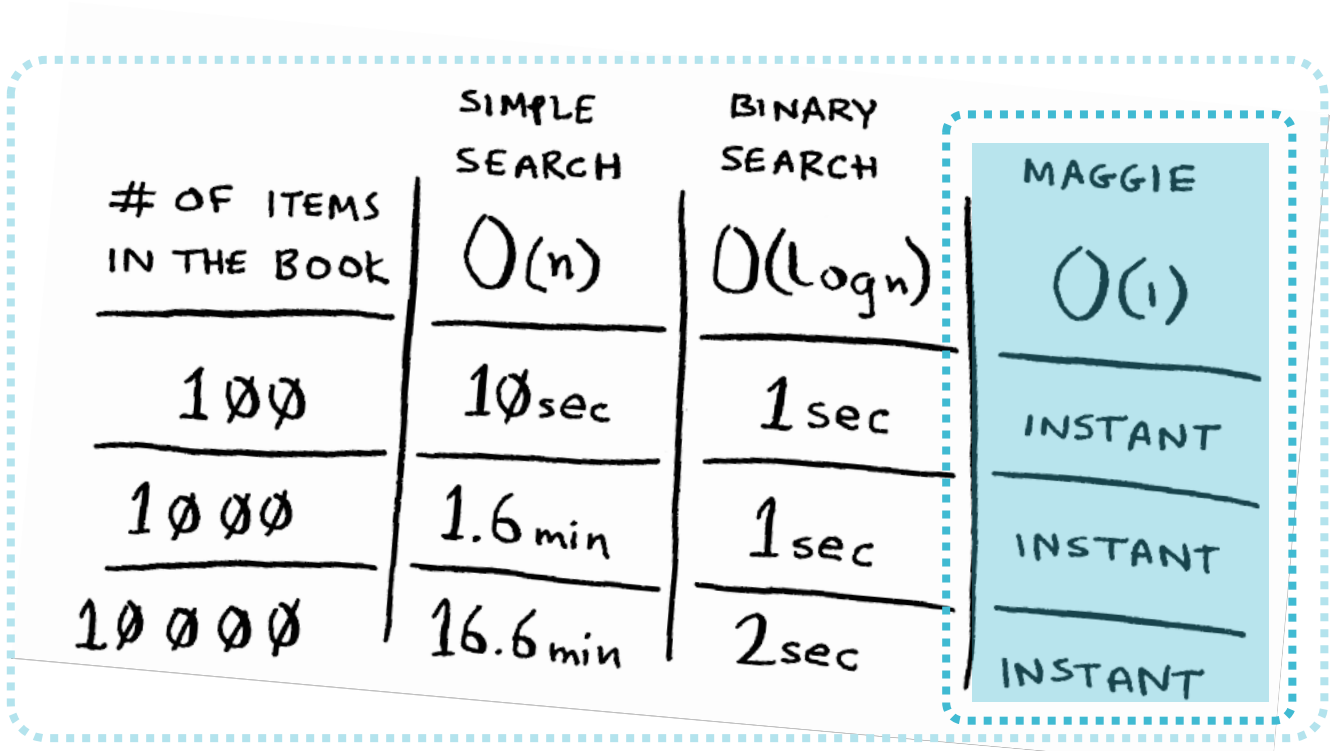
- Binary Search is fast, but **looking up in a book is a challenge**, even in a sorted book.
- What if your assistant "Maggie" can **memorize all names and prices**!
- Then, no need to look up anything, and you can **find the prices instantly**.



Hashing

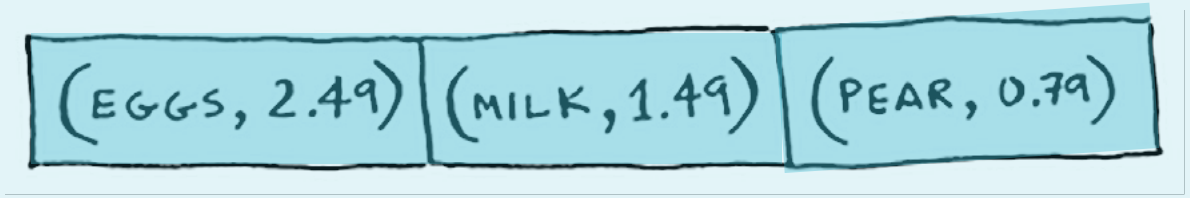
- This is wonderful:
- **Maggie** can give you the price in **O(1) time.**

- No matter which item or **how big** the pricebook is.
- Maggie will be even **faster than Binary Search**.



Hashing

- The pricebook is actually an **Array**.
- Each item in the Array contains two data:
  - **Name**
  - **Price**



- If Array is sorted by name, you can run **Binary Search** to find the price.
- But Maggie can find the price in O(1) time.
- This is basically how **Hashing** works.

Hashing

- Some **definitions** related to **Hashing:**

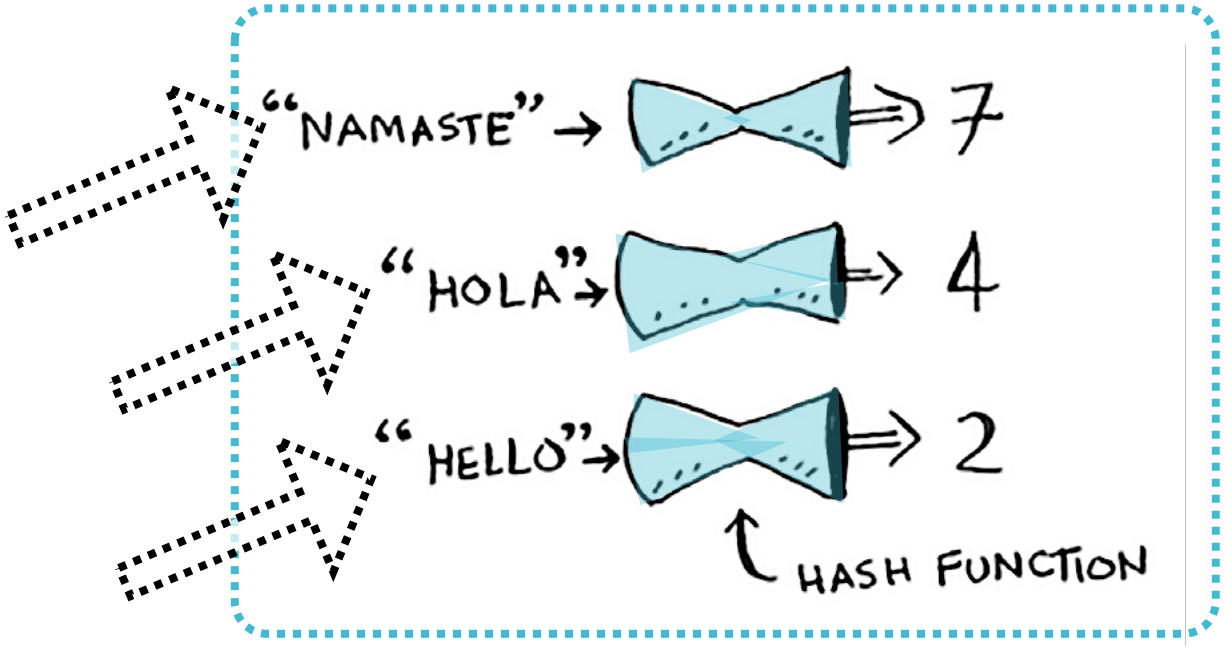
- **Hash Table:** a collection of items that are stored in a form which is **easy to find** them later.

- **Each position** of the hash table are called **a Slot** and can hold an item and is named by an integer value.

- **Hash Function:** is a function that maps an item to the slot where that item belongs in the **Hash Table**.

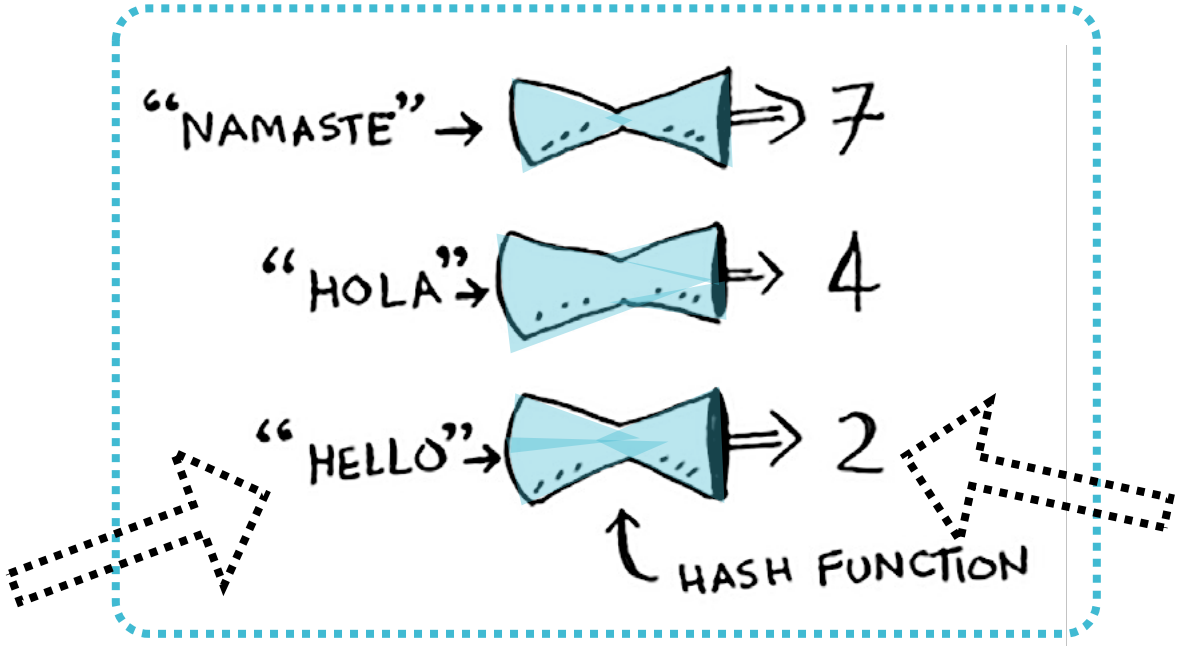
Hash Function

- **Hash function** takes an item in the collection and returns an integer in the range of slots.
- If there are ***m*** **slots**, Hash function returns an integer **between *0* and *m– 1***.



Hash Function

- **Hash function** should be **consistent**.
- For example, if you put "**Hello**", then it returns "**2**".
- All the time you put "**Hello**" it must return "**2**".



Quiz

- Which of these functions are **consistent**?

a) f(x) = 1
b) f(x) = rand()
c) f(x) = next_empty_slot_in_memory()
d) f(x) = len(x)

Answer

- Which of these functions are **consistent**?

a) f(x) = 1 **is consistent:**
   it returns always "1"

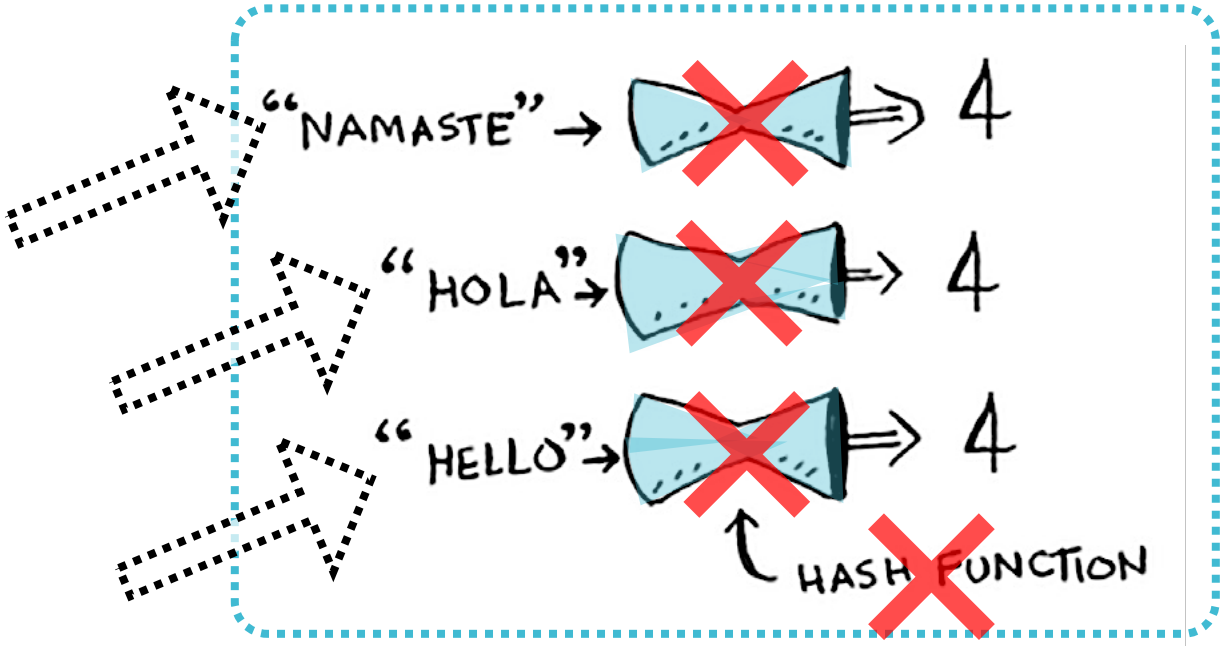
b) f(x) = rand() **is not consistent:**
   it returns a random number every time

c) f(x) = next_empty_slot_in_memory() **not consistent:**
   it returns the index of an empty slot in memory

d) f(x) = len(x) **is consistent:**
   it returns the length (of the list or string)

Hash Function

- But in addition to consistency, Hash function must map **different items to different slots**.
- Example: a hash function is **not good** if:
  - it returns "4" when input is "Hello"
  - it returns "4" when input is "Hola"
  - it returns "4" when input is "Namaste".



Hash Function

- How can we choose a good **Hash Fucntion**?
- And how does a Hash Function **works**?

- There are **different methods** for that**:**
  - Division
  - Truncation
  - Folding
  - Hashing Strings

Hash Function

- **Division:** divide the key by the size of the Hash table and take the remainder:

$$h(\mathbf{key}) = \mathbf{key} \ \% \ m$$

- **Example:** If the key is **4873152**, and the size of the hash table is **100**, then the hash value is **52**

$$4873152 \ \% \ 100 = 52$$

Hash Function

- A more sophisticated function is called **Multiply-Add-and-Divide (MAD)**:

$$h(key) = [\ (a*key + b)\ \%\ p\ ]\ \%\ m$$

- In this formula:
  - **m** is the size of the table
  - **p** is a **prime number** typically larger than m
  - a & b are random integers within interval [o, p] & a>o

Hash Function

- **MAD** function has the advantage of **eliminating the repeated patterns** in the hash codes.
- It can get closer to a **good hash function.** It is actually used for **compression**!
- The probability of **collisions** is nearly **1/N**

Quiz

- Write a has function that uses **Multiply-Add-and-Divide (MAD)** to generate hash values of the keys, with the following setting:
  - **m** = 12000
  - **p** = 109345121

- Then **call the function 3 times** and print the hash values of 2022.

- Did you **expect** the output?

Answer

- Example implementation of the function

```
from random import randrange

def hash_function(key):
    m = 12000
    p = 109345121
    a = 1 + randrange(p - 1)
    b = randrange(p)
    return (a * key + b) % p % m

print(hash_function(2022))
print(hash_function(2022))
print(hash_function(2022))
```

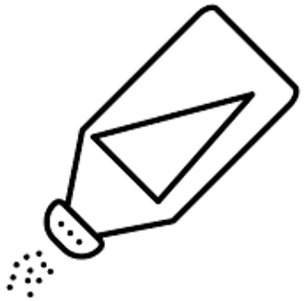
Example [Output]: 7762
10372
4421

Answer

Did you **expect** the output?

In real-world scenarios the hash values are "**salted**" with an **unpredictable** random value.

Although they **remain constant** within individual processes, they are not predictable between repeated invocations of functions.



Example [Output]: 7762
10372
4421

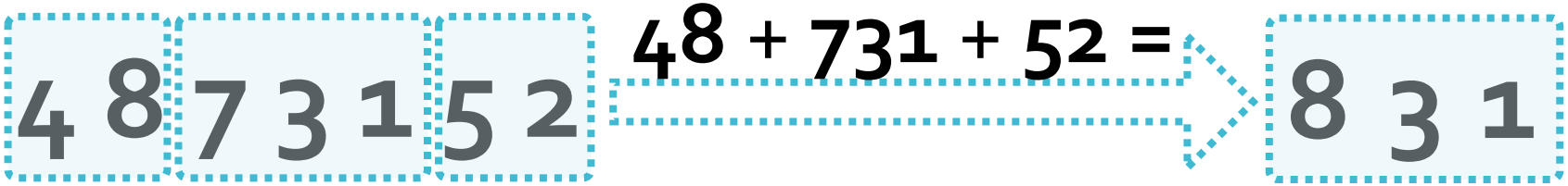
Hash Function

- **Truncation:** For large integers, some columns in the key are ignored and not used in the computation of the Hash table index

- **Example:** If the key is **4873152**, hash value is **812**.

Hash Function

- **Folding:** split a key into multiple parts & combine them into a single integer value, by adding them.
- **Example:** If the key is **4873152** the hash value **831**



Hash Function

- **Hashing Strings:** sum the **ASCII** values of the individual characters.

ASCII printable characters

32	space	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

Hash Function

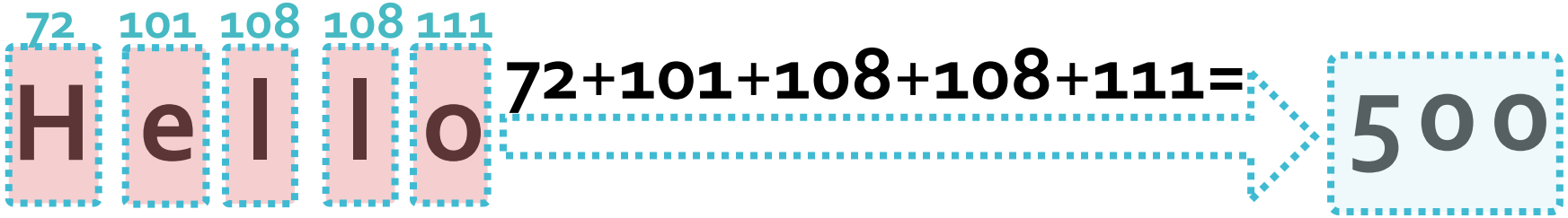
- **Hashing Strings:** sum **ASCII** values of individual characters.
- **Example:** If the key is **"Hello"** the hash value is **500**
- Why?

ASCII printable characters

32	space	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

Hash Function

- **Example:** If the key is **"Hello"** the hash value is **500**



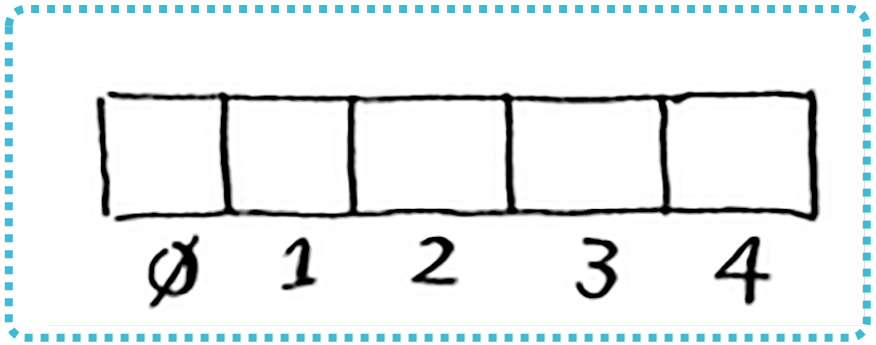
Hash Table

- If you put a hash function & an array together, you can build a new Data Structure called **Hash Table**.
- Array and List can map straight to memory, but **Hash Table is smarter.**
- Hash Table uses Hash Function to intelligently figure out **where in the memory** to store the items.

Hash Table

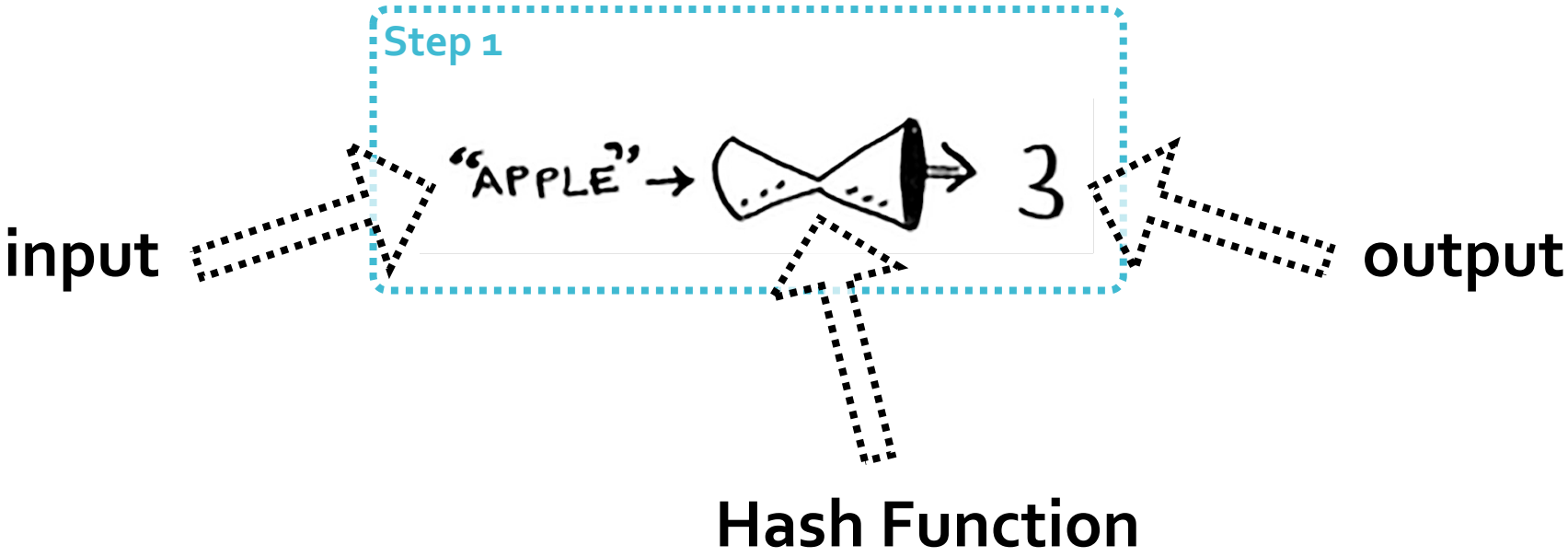
- Lets assume we chose a good Hash function.
- How can we make a **Hash Table** for Maggie's task?

- We start with an empty array:



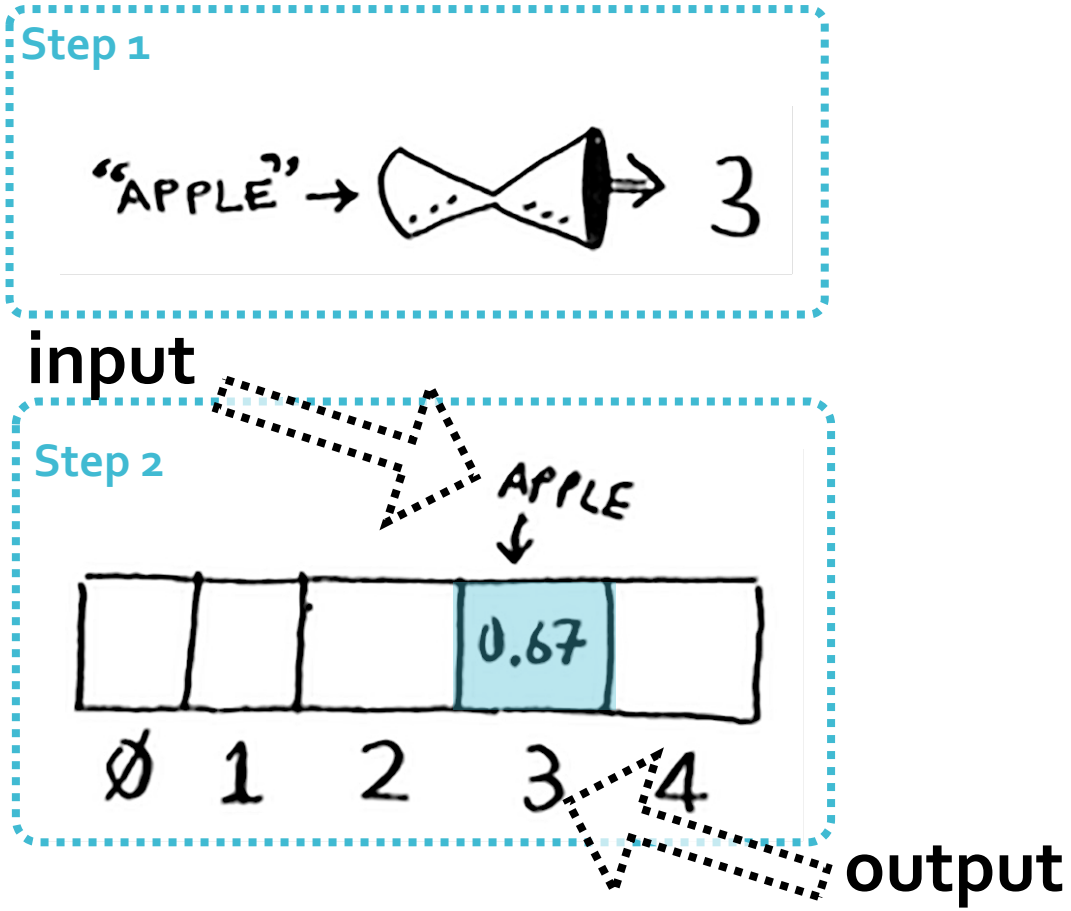
Hash Table

- We store all the prices in this array.
- Lets add **apple**, the Hash function **outputs "3".**



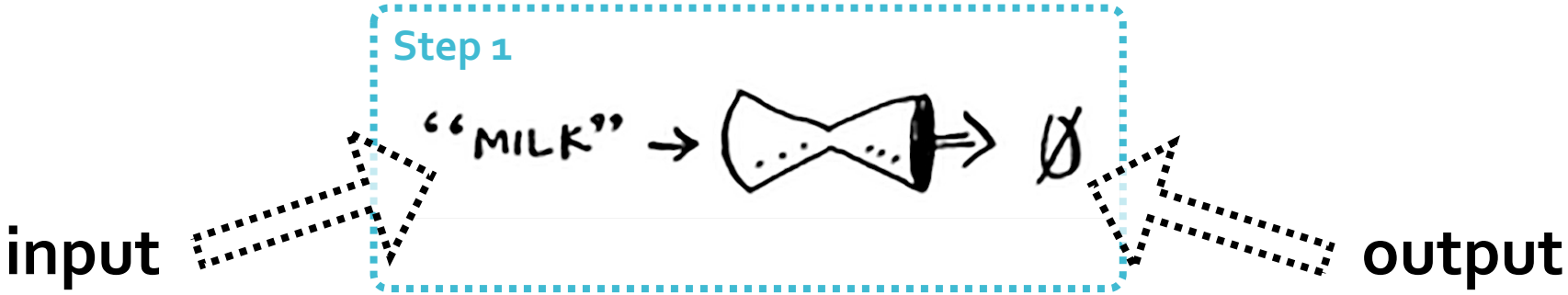
Hash Table

- We add the **price** of apple at **index 3** in the array.



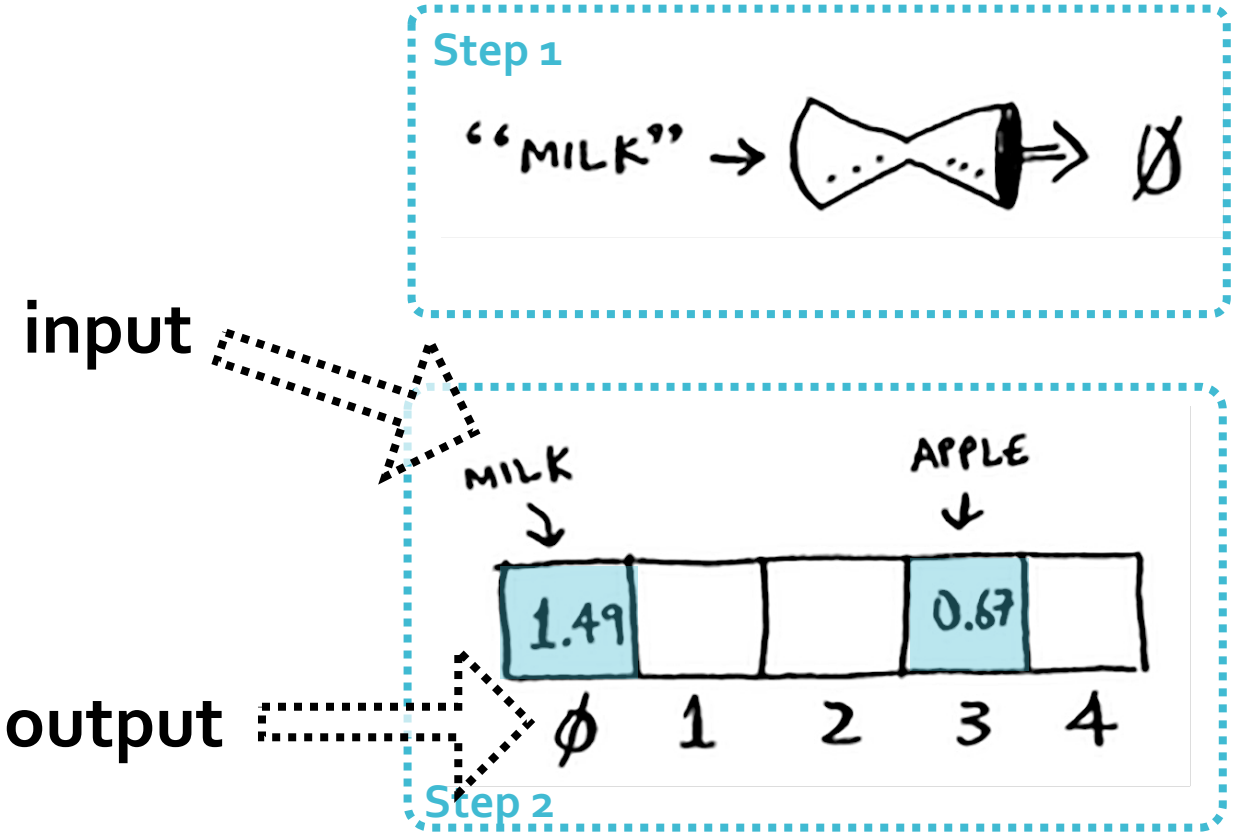
Hash Table

- Lets add **milk**, the Hash function **outputs "o".**



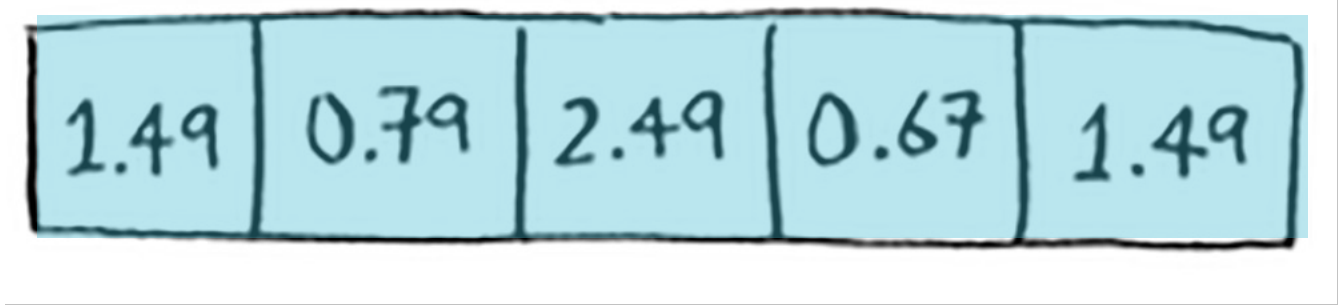
Hash Table

- Lets add **milk**, the Hash function **outputs "0".**



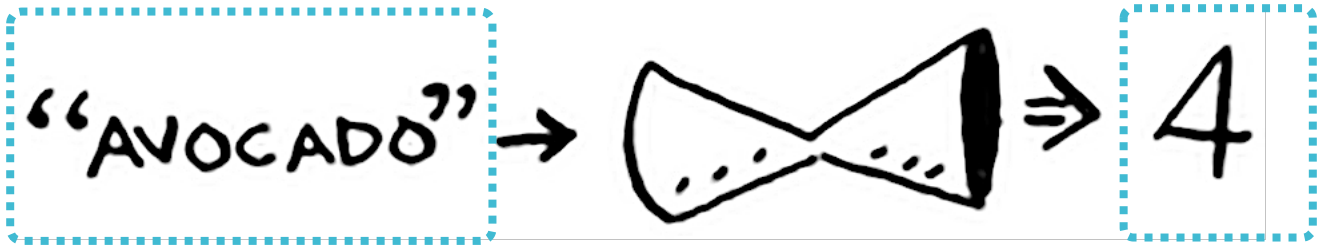
Hash Table

- Keep going, till the **whole array** will be full of prices.



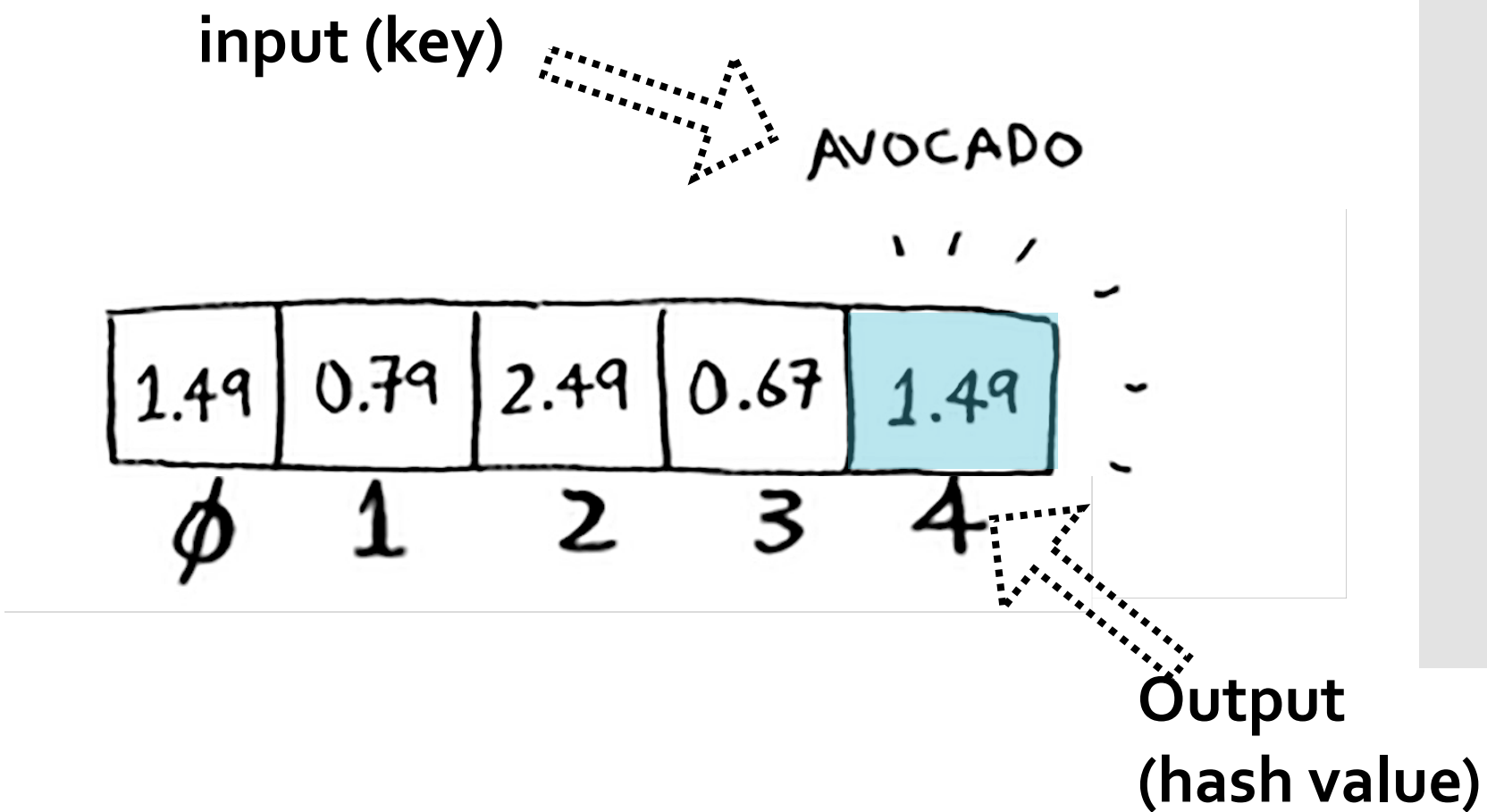
Hash Table

- Now you want to know the price of an **Avocado**:
  - **No need to wait for a Search** algorithm to do all the operations to find it.
  - Just put "Avocado" into the **Hash Function**!



Hash Table

- It will instantly tell that the **price** is stored at **index 4**.



Hash Table

- Hash Table is one of the **most useful data structure.**
- It is really fast!
- It has other names **Dictionary**, **Hash Maps**, or **Associative Array**.

Hash Table

- The **efficiency of hashing** depends in large part on the selection of a good hash function.
- **Perfect Hash Function:** given a collection of items, a hash function that maps each item into a unique slot.
- But this is **rarely achieved.**
- Instead, we can choose a **good Hash Function** that will **distribute the keys** across the range of Hash Table indices as evenly as possible.

Hash Table

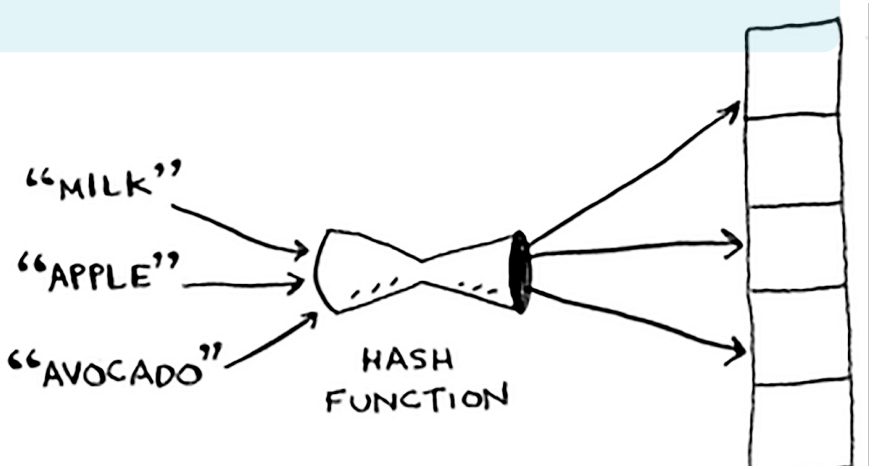
- **Guidelines** for choosing a **Hash Function**:
  - Computation should be simple to produce **quick results.**
  - Table size should be a **prime number** to produce a better distribution and fewer collisions.

Hash Function

- But we care about **performance?**
- Well…. we need to first understand what **Collisions** are.

Hashing

- Before, we assumed a hash function **always maps different keys** to **different slots** in the array.
- In **reality**, this is almost **impossible**.



- Lets check an **example**:

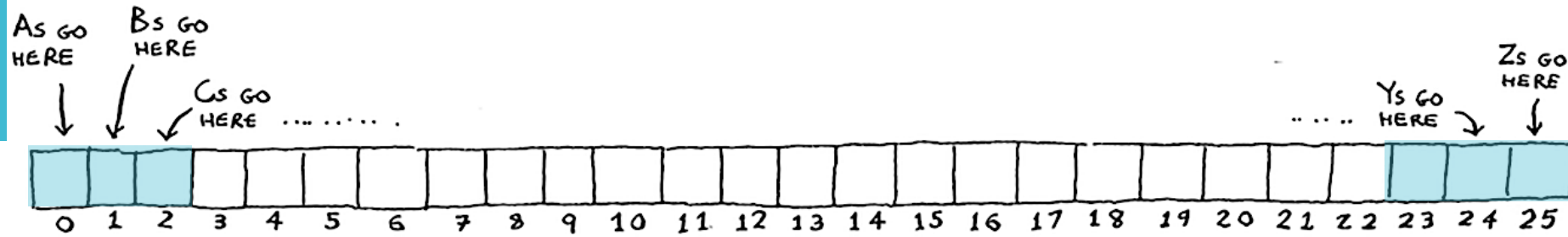
Hashing

- Suppose you have an array with **26 slots**.
- Similar to what we said previously, it stores the prices of products.

- "A" -> 0
- "B" -> 1

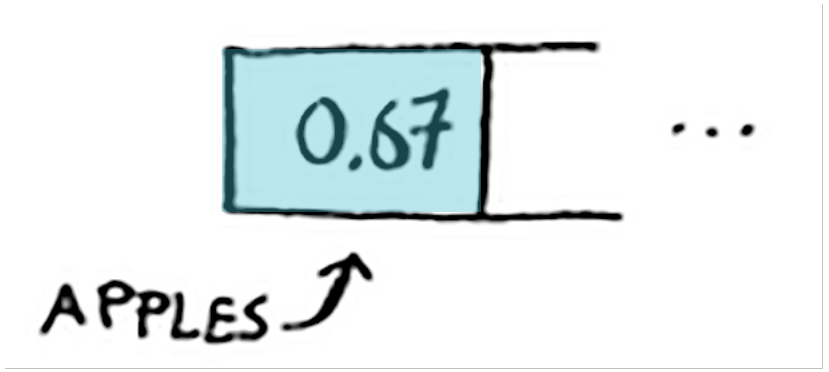
...

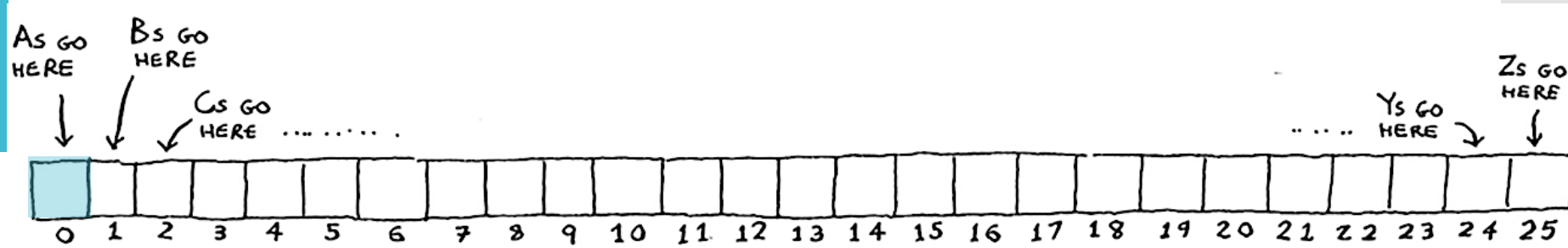
- "Z" -> 25



Hashing

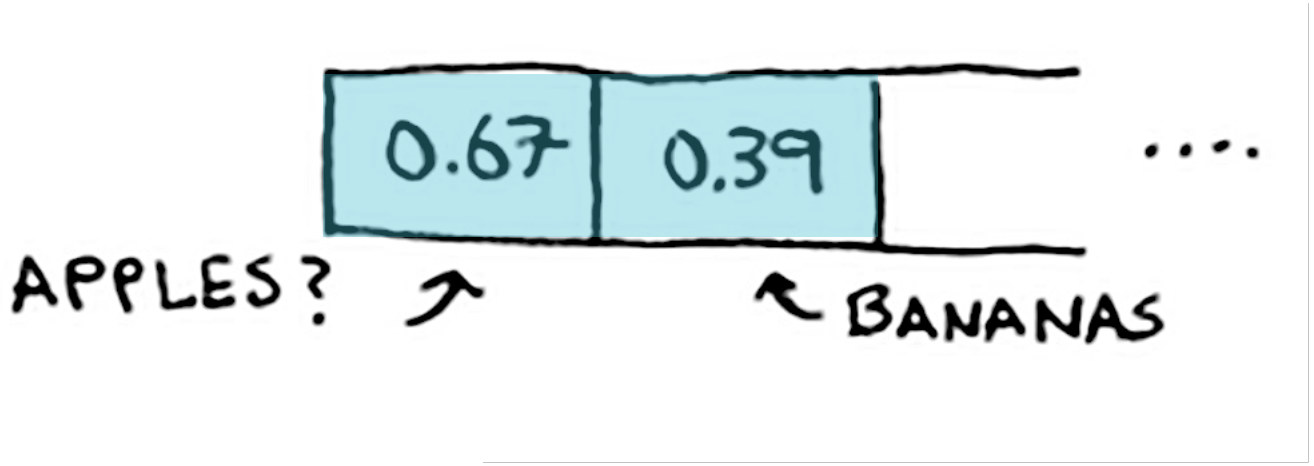
- Now, you want to put the **price of apples** in your hash. You get assigned the first slot.

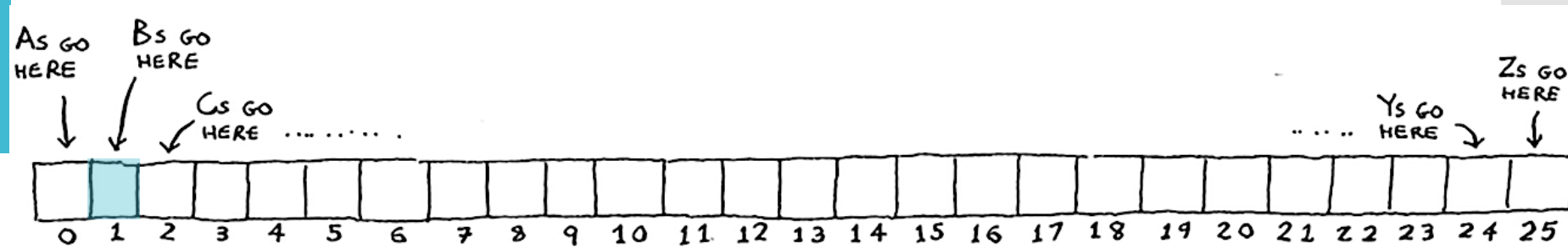




Hashing

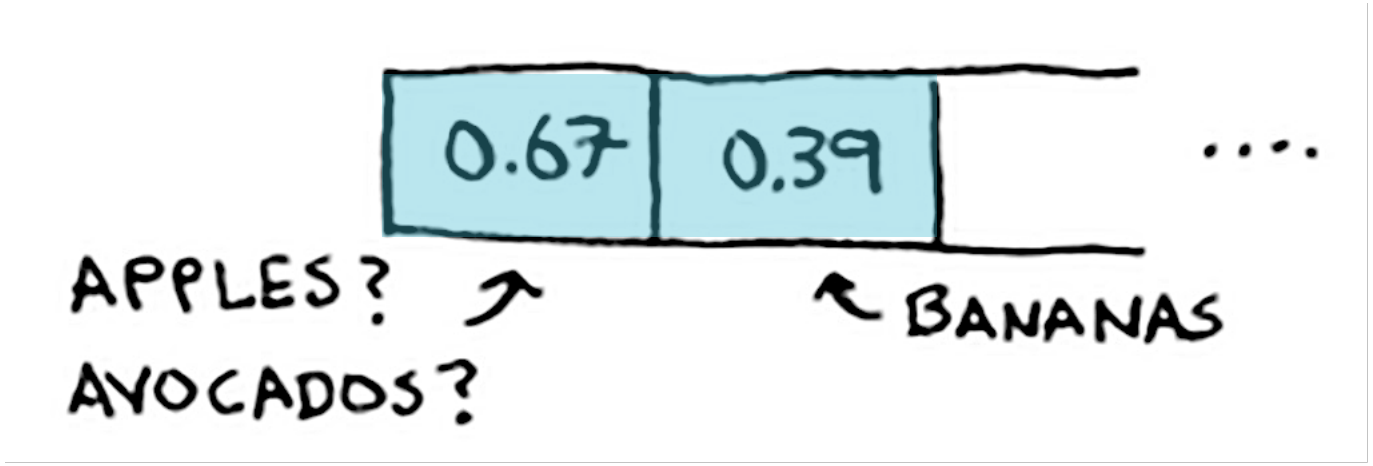
- Now you want to put the **price of banana**.
- You get assigned the second slot again!

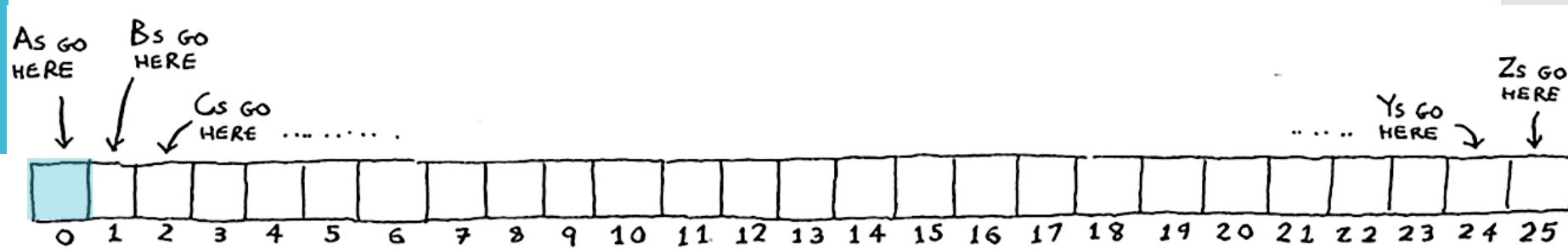




Hashing

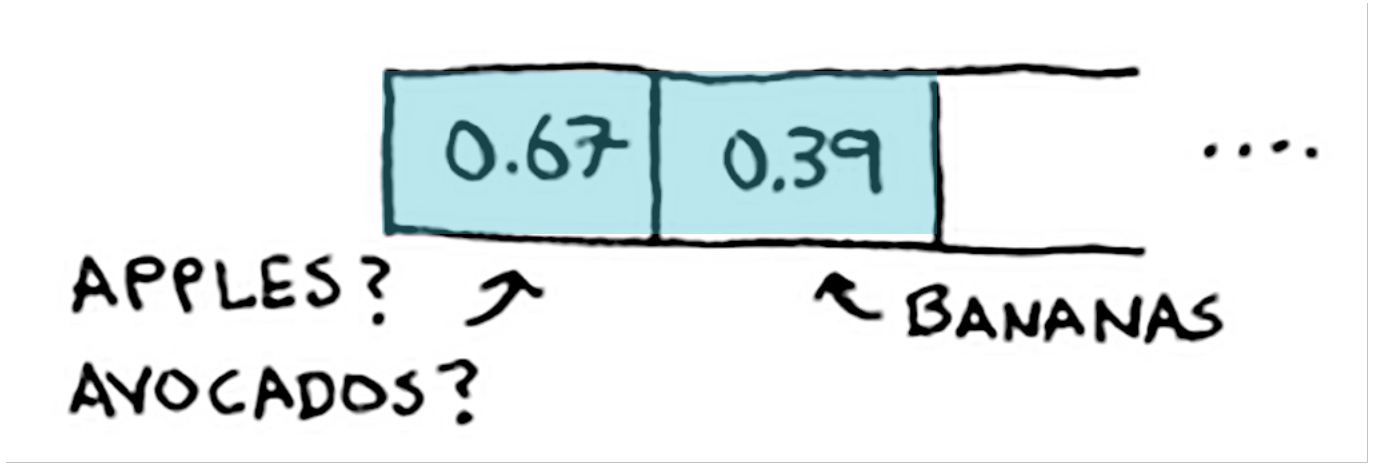
- But if you want to put the **price of avocados.**
- You get assigned the **first slot** again!

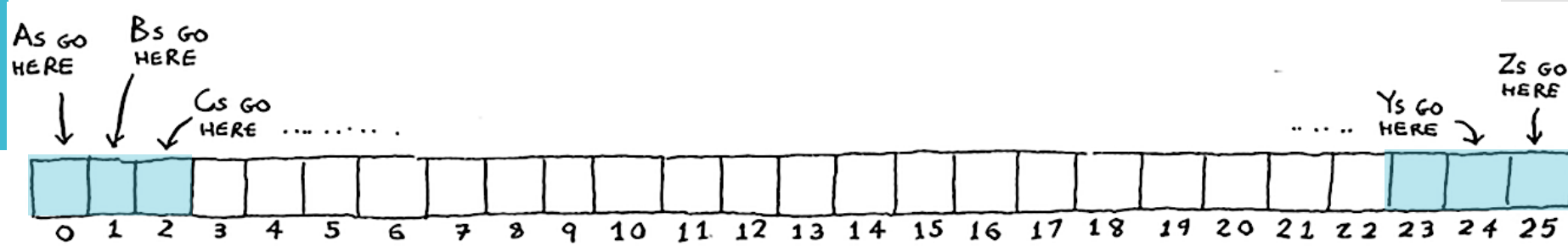




Collision

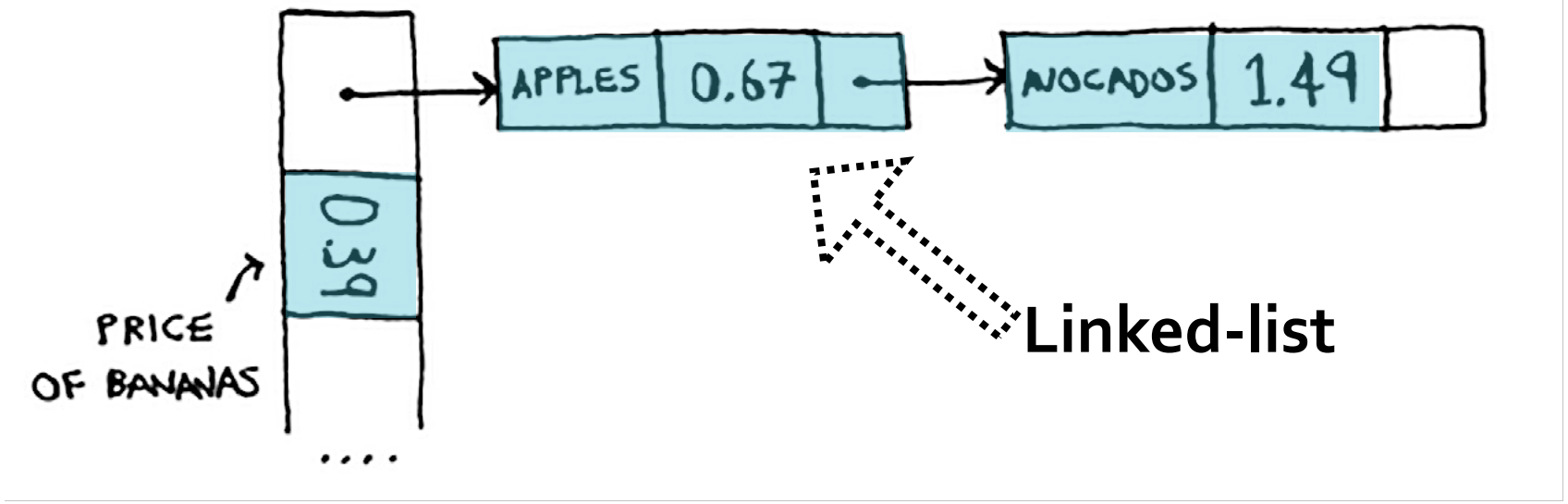
- But apples have that slot already!
- **This is called a collision**: two keys have been assigned the same slot.





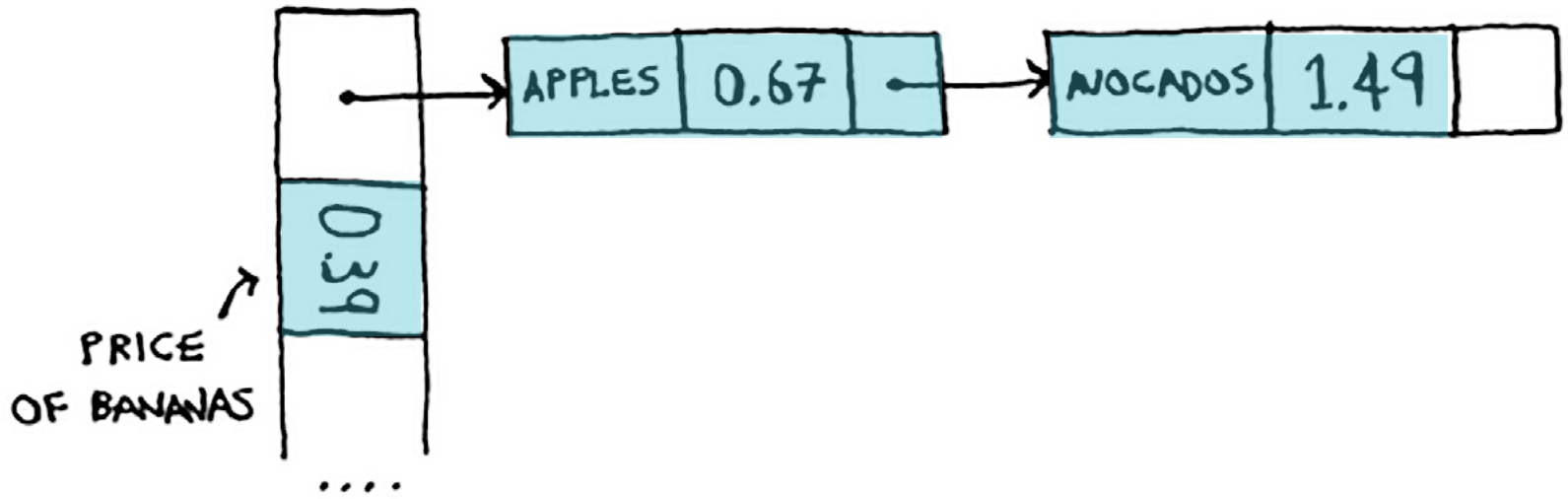
Collision

- There are different ways to deal with **collisions**.
- **The simplest one is this:** if multiple keys map to the same slot, start a **Linked-list** at that slot.



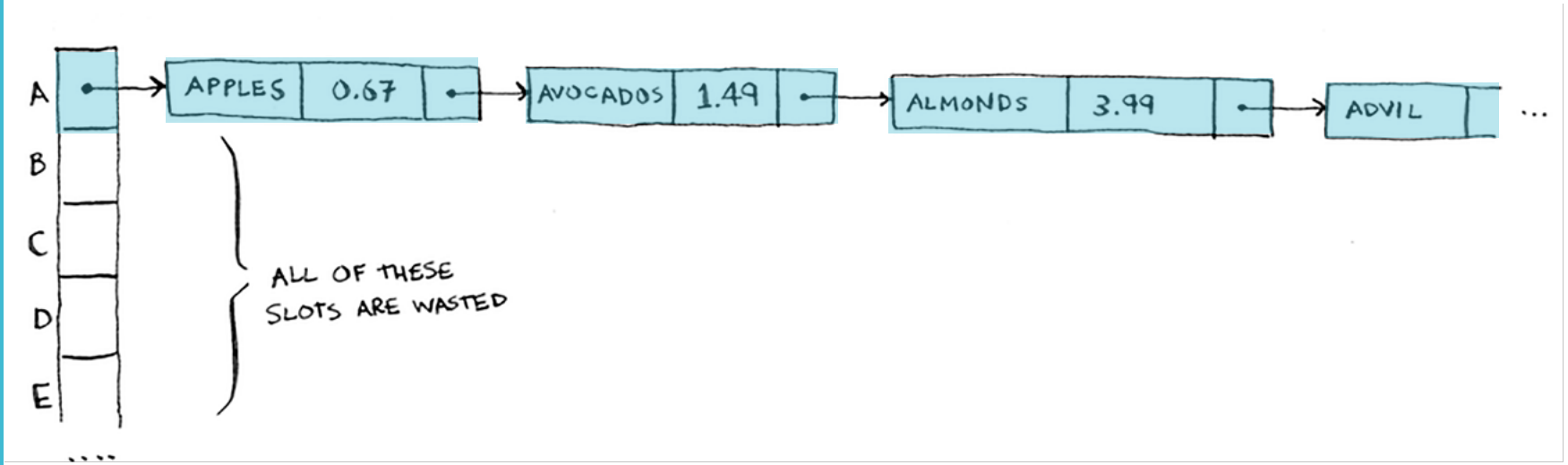
Collision

- If you need to know:
  - the price of **bananas**, it's still **quick**.
  - the price of **apples** or **avocados**, it's a **slower**.



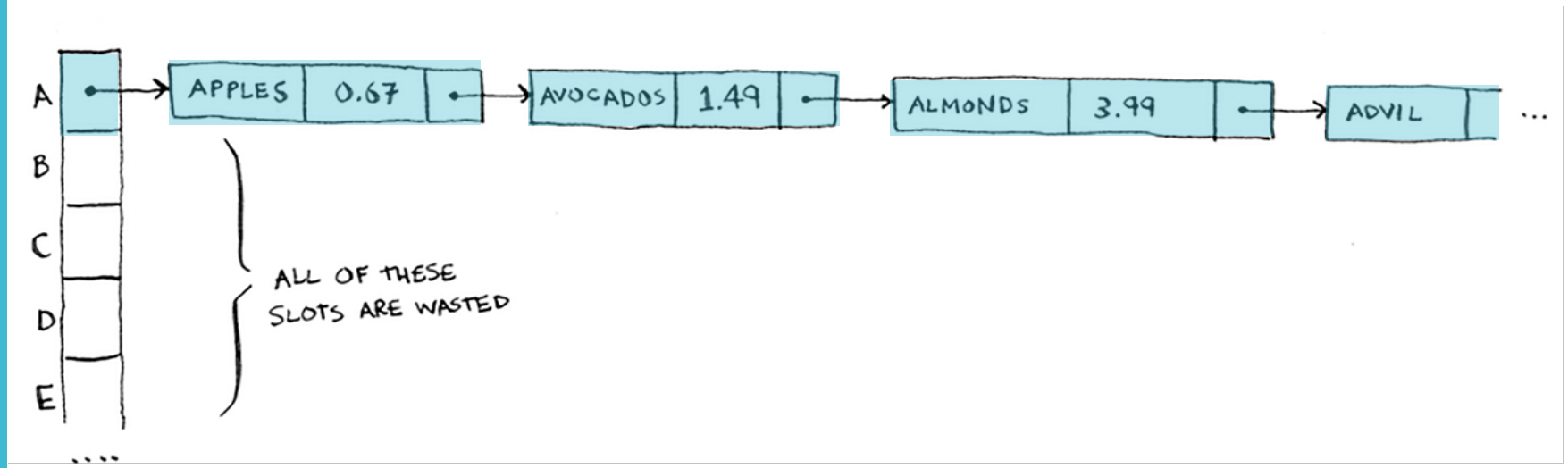
Collision

- If the **Linked-list is small, no big deal**.
- But suppose you work at a grocery store where you only sell **a lot of products that starts with the letter A**.



Collision

- Then Hash Table is totally empty **except for 1 slot!**
- And that slot has a **giant Linked-list**!
- Every item in this Hash table is in the Linked-list.
- It's going to **slow down** your hash table.



Collision

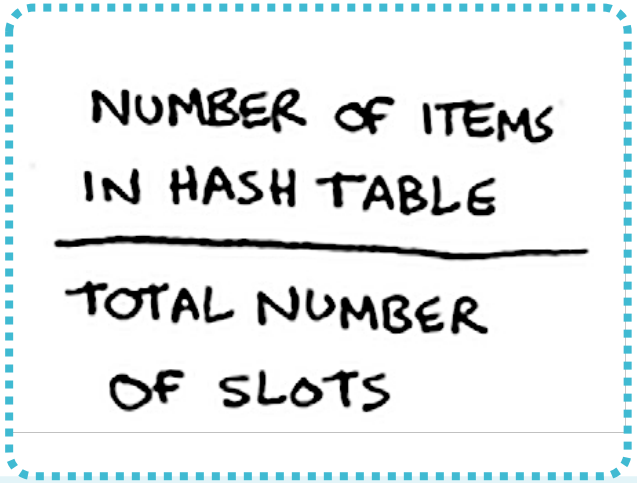
- **Three lessons** here:
  - Such long **Linked-lists slows down** a lot a hash table.
  - Hash function should **map** keys **evenly** all over hashes.
  - **Good Hash function** should give a **very few collisions**.

Collision

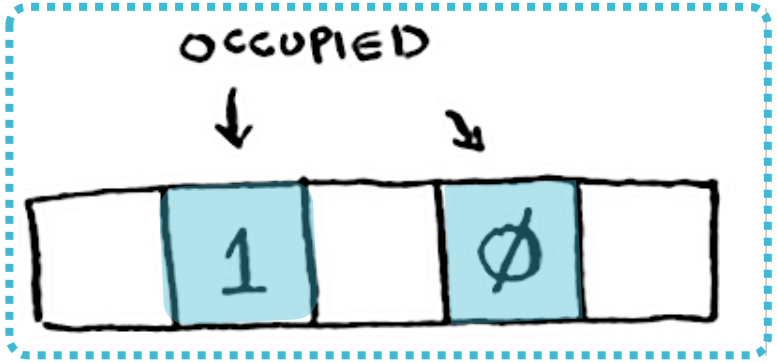
- **Open Addressing:** process of resolving collisions by looking into the Hash table and trying to find another open slot.
- **Example:** start at the original hash value position and then move in a sequential manner through the slots until finding an empty slot that is empty.
- Visiting each slot one at a time, with open addressing technique, is called **Linear Probing**.

Performance

- **Load factor** of a Hash Table is an important factor.
- It is calculated by the **number of items** divided by the total **number of slots** in Hash Table.

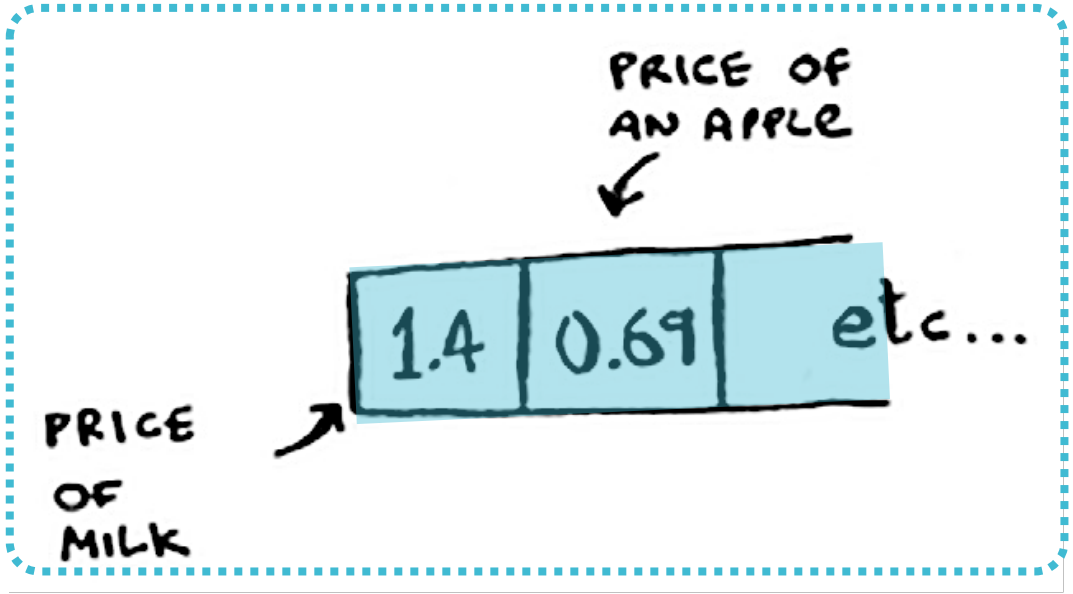


- **Example:** this Hash table has a load factor of **2/5**



Performance

- Suppose you need to store the **price of 100 items** in your hash table.
- Your hash table has **100 slots**, and hence in the best case, **each item** will **get its own slot**.
- This hash table has a **load factor of 1**.

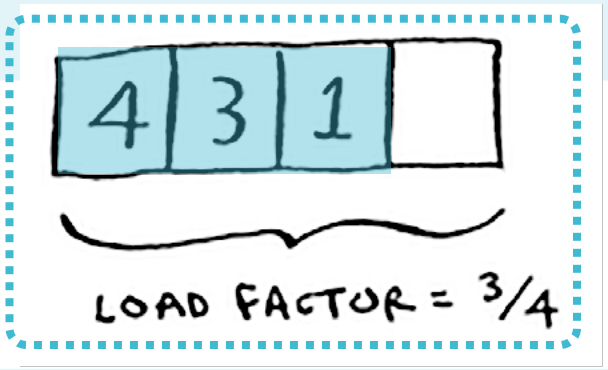


Performance

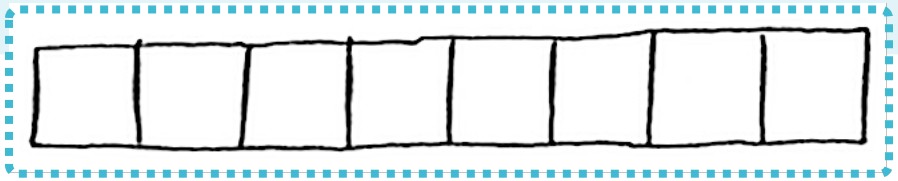
- What if your hash table has only 50 slots? Then it has a **load factor of 2**.
- Having a **load factor greater than 1** means you have more items than slots in your array.
- **Resizing:** when the **load factor grows**, more slots to have to be added to the hash table.

Performance

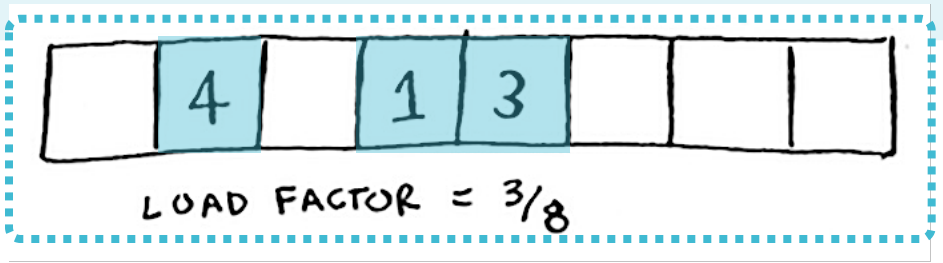
- **Example:** suppose you have this hash table that is getting pretty full.



- You need to **resize** this hash table:
  1) create a new array that's bigger (e.g., twice the size)

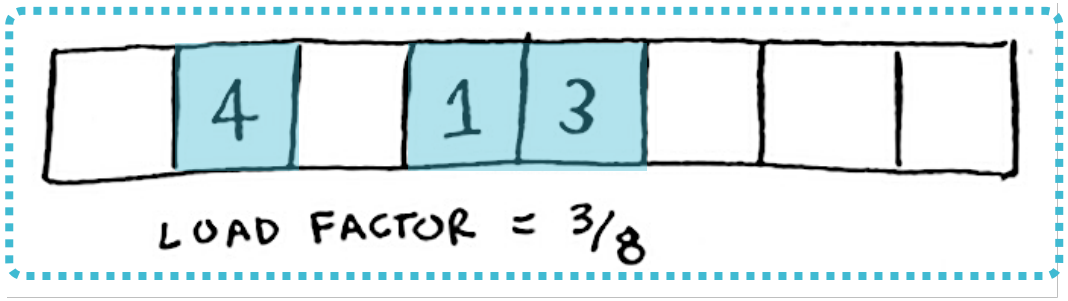


  2) then, **re-insert** all items into this new Hash table using the hash function:



Performance

- This new table has a load factor of 3/8: **Much better!**



- **Lower load factor** means **fewer collisions** the Hash table will **perform better**.

- **Rule of thumb**: resize when load factor is greater than **0.7**

Performance

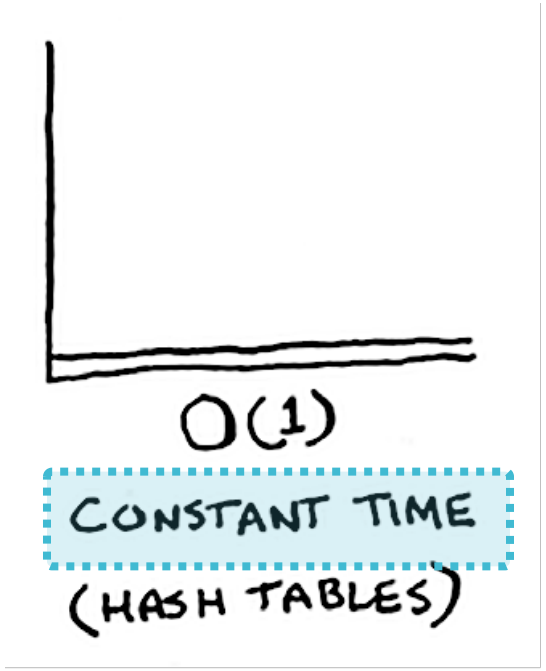
- **Resizing is expensive** and takes a lot of time.
- Resizing **should not occur too often**.
- But still, by averaged, **Hash tables take O(1)** even with resizing.

Performance

- So what is the conclusion?
  - hash tables are still **really fast:** average case is O(1) (constant time).
  - O(1) means the **time** taken is **the same**, no matter **how big** is the hash table.

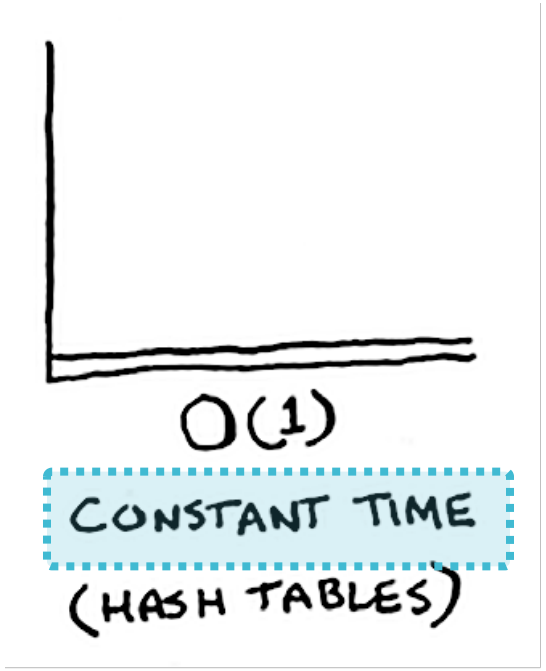
Performance

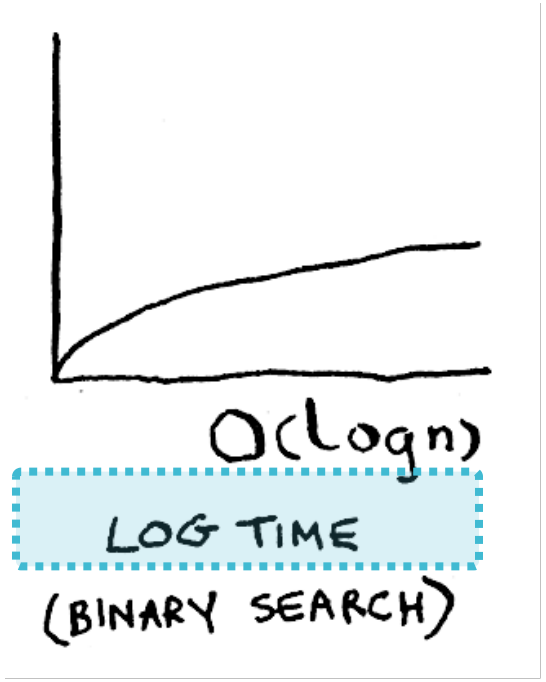
- Hash Tables



Performance

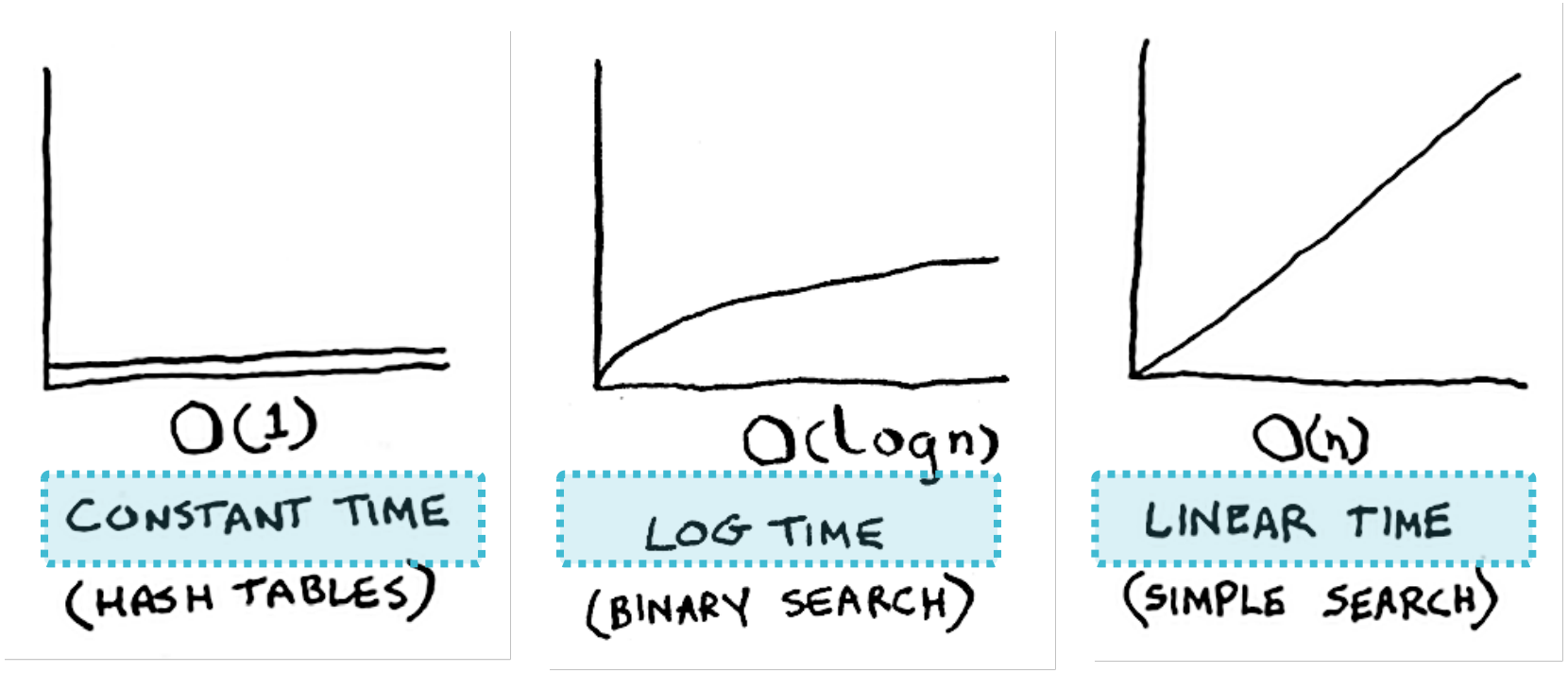
- **Hash Tables** vs **Binary Search**





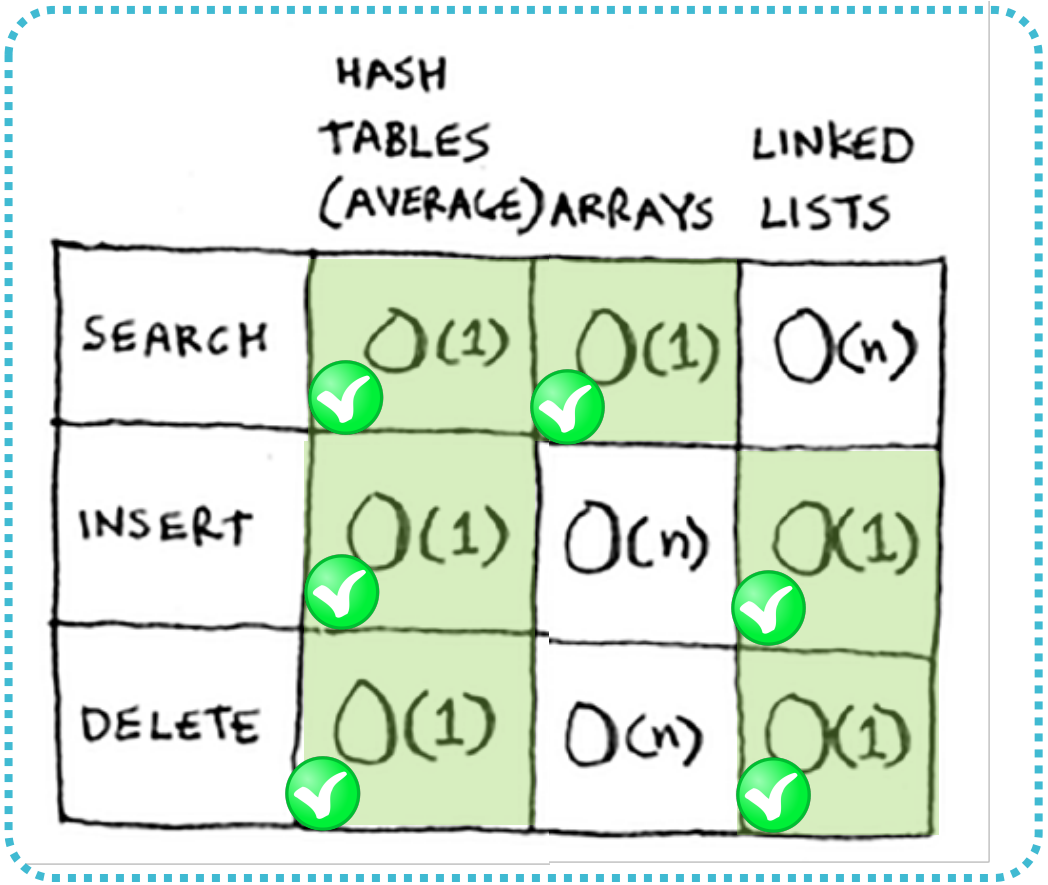
Performance

- **Hash Tables** vs **Binary Search** vs **Simple Search**



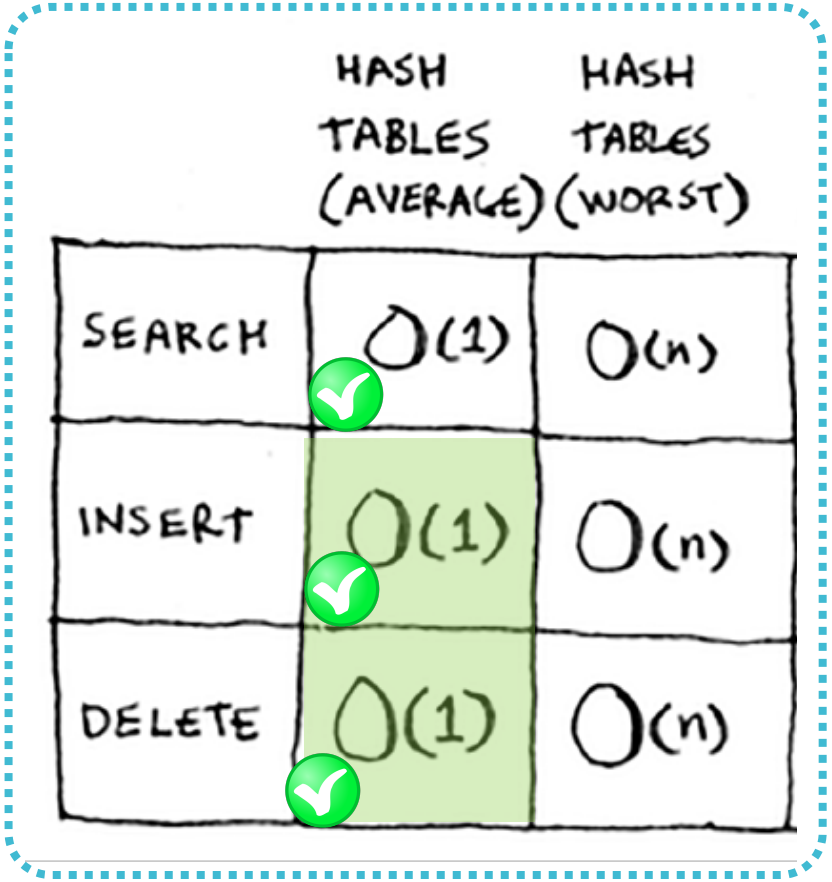
Performance

- **Hash Tables** vs **Binary Search** vs **Simple Search**



Performance

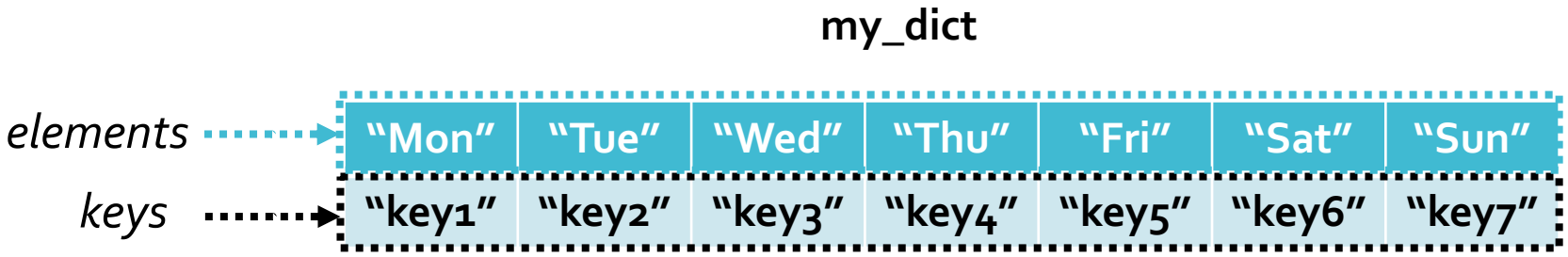
- **Hash tables (in average)** are **O(1):**
  - as fast as arrays at searching (getting a value at an index).
  - as fast as linked lists at insert and delete.
- **Hash tables (in worst case)** are **O(n)** which is slow at all of those.



Hashing

- Python and other languages have **built-in hash tables** and no need to implement the hash tables.

Remember?



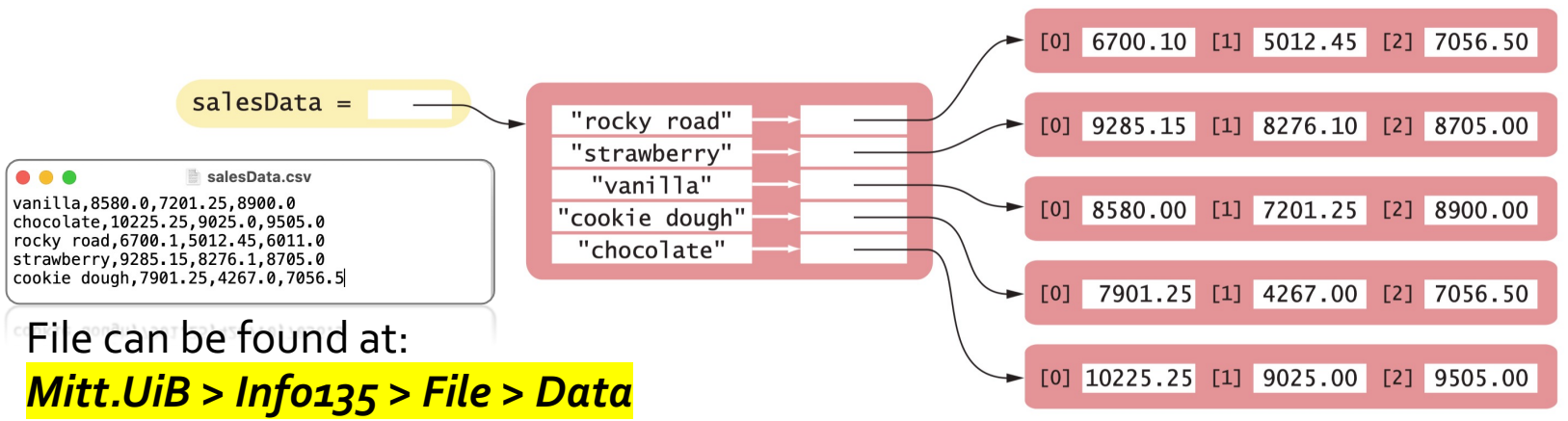
- Data collection that is **unordered**, **mutable**, and **indexed**
- Can have **duplicates**
- Represented with **curly brackets** and specifying the **key & value** pairs {key:value}

- Example:

```
my_dict =
{"key1":"Mon","key2":"Tue","key3":"Wed",
 "key4":"Thu","key5":"Fri","key6":"Sat",
 "key7":"Sun"}
```

Quiz

- Write a Python program **`build_dict_from_data()`** that receives the name of file as input parameter and goes through and reads it.
- Use the program to read a file called **salesData.csv**, that contains a collection of sales for an ice-cream shop.
- The program should create and print a Python dictionary where the **keys** are ice-cream **flavors** and the **values** are **lists** of numbers representing the sales of the shop within different months.



salesData.csv

```
vanilla,8580.0,7201.25,8900.0
chocolate,10225.25,9025.0,9505.0
rocky road,6700.1,5012.45,6011.0
strawberry,9285.15,8276.1,8705.0
cookie dough,7901.25,4267.0,7056.5
```

File can be found at:

Mitt.UiB > Info135 > File > Data

Quiz

Reminder for opening files:

- **Syntax**:

```
with open(“filename” , "r") as file:
    data = file.read()
```

- **Example**:

```
with open(“Hello.txt”) as file:
    data = file.read()
```

Answer

- Reading the and building a dictionary for **every row of the file** (key:values)

Part 1

```
def build_dict_from_date(filename):
    salesData = {}

    with open(filename, "r") as infile:
        for line in infile:
            fields = line.split(",")
            flavor = fields[0]
            salesData[flavor] = make_list(fields)

    return salesData
```

Answer

- **Generating list** of numbers (monthly sales of the shop).

Part 2

```
def make_list(fields):
    storeSales = []
    for i in range(1, len(fields)):
        sales = float(fields[i])
        storeSales.append(sales)

    return storeSales
```

```
salesData = build_dict_from_date("salesData.csv")

for key, value in salesData.items():
    print(key, ' : ', value)
```

Answer

❖ **Generating list** of numbers (monthly sales of the shop).

```
def make_list(fields):
    storeSales = []
    for i in range(1, len(fields)):
        sales = float(fields[i])
        storeSales.append(sales)

    return storeSales
```

```
salesData = build_dict_from_date("salesData.csv")

for key, value in salesData.items():
    print(key, ' : ', value)
```

Part 3

[Output]:

```
vanilla  :  [8580.0, 7201.25, 8900.0]
chocolate  :  [10225.25, 9025.0, 9505.0]
rocky road  :  [6700.1, 5012.45, 6011.0]
strawberry  :  [9285.15, 8276.1, 8705.0]
cookie dough  :  [7901.25, 4267.0, 7056.5]
```

Hashing

- Python has also a built in Hash function:

  `hash(`*`object`*`)`

- It return hash value of the object. You can also **override** it!

- Numbers that equal value (even with different types) have the same hash value.

- **Example of usage**: to quickly compare dictionary keys during a dictionary lookup.

Hashing

Hashing

```
class Car:
    def __init__(self, model, brand):
        self.model = model
        self.brand = brand

    def __eq__(self, other):
        return self.model == other.model \
               and self.brand == other.brand

    def __hash__(self):
        return self.model \
               * ord(self.brand[0])\
               * ord(self.brand[1])\
               * ord(self.brand[2])\
               * ord(self.brand[3])
```

Hashing

❖Hashing

```
my_car1 = Car(2020, 'Benz')
my_car2 = Car(2019, 'Audi')
print("My Hash value for car 1:", hash(my_car1))
print("My Hash value for car 2:", hash(my_car2))
```

[Output:]

```
My Hash value for car 1: 180704594400
My Hash value for car 2: 161222197500
```

Is this a good hash function? why?

Hashing

❖Hashing

```
my_car1 = Car(2020, 'Benz')
my_car2 = Car(2019, 'Audi')
print("My Hash value for car 1:", hash(my_car1))
print("My Hash value for car 2:", hash(my_car2))
```

[Output:]

```
My Hash value for car 1: 180704594400
My Hash value for car 2: 161222197500
```

Is this a good hash function? why?
No. There will be collisions when the cars have **the same brand and model**. This has not resolved.

Quiz

1) Write a class **Employee** where you override Hash function. Define **your own** Hash function which works based on the **`id_number`** of an employee.

```
class Employee:
    def __init__(self, id_number):
        
    def __eq__(self, other):
        
    def __hash__(self):
        
```

Quiz

2) In Python only **immutable objects can be hashed**. But why?

Answer

- Hash function based on **`id_number`** of employees.

```
class Employee:
    def __init__(self, id_number):
        self.id_number = id_number

    def __eq__(self, other):
        return self.id_number == other.id_number

    def __hash__(self):
        return hash(self.id_number)
```

```
employee1 = Employee(30122000)
employee2 = Employee(29121999)
print("Hash value for employee 1:", hash(employee1))
print("Hash value for employee 2:", hash(employee2))
```

[Output:]

```
Hash value for employee 1: 30122000
Hash value for employee 2: 29121999
```

Answer

- If a mutable objects are also allowed to be hashed, then we must **update Hash Table** every time the the **objects are changed**.
- This may mean that the object must be moved to a totally different location and this can be an **expensive operation**.

Hashing

- Popular Hash functions:
  - **Message Digest algorithm 5 (MD5)**
  - **First Published:** April 1992
  - **By** Computer Scientist **Ronald Rivest**

- **MD5** hash is created by taking a **string of any length** and encoding it into a **128-bit** fingerprint.

=

```
79054025
255fb1a2
6e4bc422
aef54eb4
```

Not recommended anymore!

- **Demo:** www.md5hashgenerator.com

Hashing

- A popular Hash function:
  - **Secure Hash Algorithms (SHA)**
  - **First Published:** April 1993 (SHA-0)
  - **By:** National Institute of Standards and Tech
- The message generated by SHA ranges from **160 to 512 bits.**
- **Demo:** xorbin.com/tools/sha256-hash-calculator

Hashing

- One of the popular applications of these algorithms is **password hashing**.

Hash Application

ref: youtu.be/cczlpiiu42M

Hashing

- Lets try hashing with this algorithms!

Hashing

- This is a function that uses **SHA hashing**

```
import hashlib as hl
```

```
def hash_it_by_sha1(user_str):
    hash_str = hl.sha1(user_str.encode()).hexdigest()
    return hash_str
```

```
print('1) Hash by sha:', hash_it_by_sha1('Hello Bergen'))
print('2) Hash by sha:', hash_it_by_sha1('Hello Bergen!'))
print('3) Hash by sha:', hash_it_by_sha1('Hello Bergen!!'))
```

[Output:]

```
1) Hash by sha: a527128b69070db7c29dbe7694c57db1898c1a2e
2) Hash by sha: 054a328b3889660a1e000a5c6938f6c97a02ba47
3) Hash by sha: a875245dfceb3728af7f2fcd7fefe490039a8527
```

Hashing

- SHA object has the following methods:
  - **`digest()`** returns the digest of the strings passed to the update method so far (in byte format where it may contain non-ASCII characters).
  - **`hexdigest()`** is similar to **`digest()`** except this returns hexadecimal digits.
  - **`update(`*`arg`*`)`** updates the object with the string *arg*.
  - **`copy()`** return a copy ("clone") of the object. This can be used to efficiently compute the digests of strings that share a common initial substring.

Hashing

Lets write a **Password class**

```
import hashlib as hl

class Password:

    def __init__(self):
        user_pass = input('Please enter your password: ')
        self.hash_pass = self.hash_it(user_pass)
```

Hashing

- Lets write a **Password class**

```
import hashlib as hl
```

```
def hash_it(self, user_pass):
    self.hash_pass = hl.sha1(user_pass.encode()).hexdigest()
    return self.hash_pass
```

Hashing

- Lets write a **Password class**

```
import hashlib as hl
```

```
def print_it(self):
    print('Your hashed password is:', self.hash_pass)
    print('Size of hash password is:', len(self.hash_pass))
```

Hashing

```
import hashlib as hl

class Password:

    def __init__(self):
        user_pass = input('Please enter your password: ')
        self.hash_pass = self.hash_it(user_pass)

    def hash_it(self, user_pass):
        self.hash_pass = hl.sha1(user_pass.encode()).hexdigest()
        return self.hash_pass

    def print_it(self):
        print('Your hashed password is:', self.hash_pass)
        print('Size of hash password is:', len(self.hash_pass))

my_password = Password()
my_password.print_it()
```

[Output:]

```
Please enter your password: bergen2020
Your hashed password is: e17b5c9a9735e58c0ba6b8974c7721428c7f23fb
Length of hashed password is: 40
```

Quiz

- In Password class, write a new method called **`log_in()`** that checks and prints if a **password**, entered by user, is **correct or not**!

```
class Password:

    def __init__(self):...

    def hash_it(self, user_pass):...

    def print_it(self):...

    def log_in(self):
```

Quiz

```
class Password:

    def __init__(self):...

    def hash_it(self, user_pass):...

    def print_it(self):...

    def log_in(self):
```

```
my_password = Password()
my_password.log_in()
```

[Output:]

```
Please enter your password: bergen2020
To login, please re-enter your password: bergen2020
Password is correct.
```

Answer

```
class Password:

    def __init__(self):...

    def hash_it(self, user_pass):...

    def print_it(self):...

    def log_in(self):
        entered_pass = input('To login,re-enter password:')
        if self.hash_pass == self.hash_it(entered_pass):
            print('Password is correct.')
        else:
            print('Sorry, password not correct.')
```

```
my_password = Password()
my_password.log_in()
```

[Output:]

```
Please enter your password: bergen2020
To login, please re-enter your password: bergen2020
Password is correct.
```

Next Lesson

- **Graphs**