# DS5010

## Intro to programming for Data Science

LECTURE 6

# TODAY

- review session 5
- PART I
  - more on classes and objects
  - inheritance
- PART II: basic data structures
  - Stack
  - Queue
  - LinkedList
  - HashTable

# REVIEW

- what are objects
- classes
- object oriented programming

# PART I

# IMPLEMENTING THE CLASS vs USING THE CLASS

- write code from two different perspectives

**implementing** a new object type with a class

- **define** the class
- define **data attributes**
  (WHAT IS the object)
- define **methods**
  (HOW TO use the object)

**using** the new object type in code

- create **instances** of the object type
- do **operations** with them

# CLASS DEFINITION OF AN OBJECT TYPE vs INSTANCE OF A CLASS

- class name is the **type**

```
class Coordinate:
```

- class is defined generically
  - use `self` to refer to some instance while defining the class
    `(self.x-self.y)**2`
  - `self` is a parameter to methods in class definition
- class defines data and methods **common across all instances**

- instance is **one specific** object

```
coord= Coordinate(1,2)
```

- data attribute values vary between instances

```
c1 = Coordinate(1,2)
c2 = Coordinate(3,4)
```

  - `c1` and `c2` have different data attribute values `c1.x` and `c2.x` because they are different objects

- instance has the **structure of the class**

# GETTER AND SETTER METHODS

```
class Animal:
    def __init__(self, age):
        self.age = age
        self.name = None
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name

    def set_age(self, newage):
        self.age = newage
    def set_name(self, newname=""):
        self.name = newname
    def __str__(self):
        return "animal:"+str(self.name)+":"+str(self.age)
```

getter

setter

**getters and setters** should be used outside of class to access data attributes

# AN INSTANCE and DOT NOTATION (RECAP)

- instantiation creates an **instance of an object**

```
a = Animal(3)
```

- **dot notation** used to access attributes (data and methods)
though it is better to use getters and setters
to access data attributes

```
a.age
```

-access data attribute
-allowed, but not recommended

```
a.get_age()
```

-access method
-best to use getters
and setters

# INFORMATION HIDING

- author of class definition may **change data attribute** variable names

replaced `age` data attribute by `years`

```
class Animal:
    def __init__(self, age):
        self.years = age
    def get_age(self):
        return self.years
```

- if you are **accessing data attributes** outside the class and class **definition changes**, may get errors
- outside of class, use getters and setters instead use `a.get_age()` NOT `a.age`
  - good style
  - easy to maintain code
  - prevents bugs

# PYTHON NOT GREAT AT INFORMATION HIDING

- allows you to **access data** from outside class definition

```
print(a.age)
```

- allows you to **write to data** from outside class definition

```
a.age = 'infinite'
```

- allows you to **create data attributes** for an instance from outside class definition

```
a.size = "tiny"
```

- it's **not good style** to do any of these!

# DEFAULT ARGUMENTS

- **default arguments** for formal parameters are used if no actual argument is given

```
def set_name(self, newname=""):
    self.name = newname
```

- default argument used here

```
a = Animal(3)
a.set_name()
print(a.get_name())
```

prints ""

- argument passed in is used here

```
a = Animal(3)
a.set_name("fluffy")
print(a.get_name())
```

prints "fluffy"

# HIERARCHIES

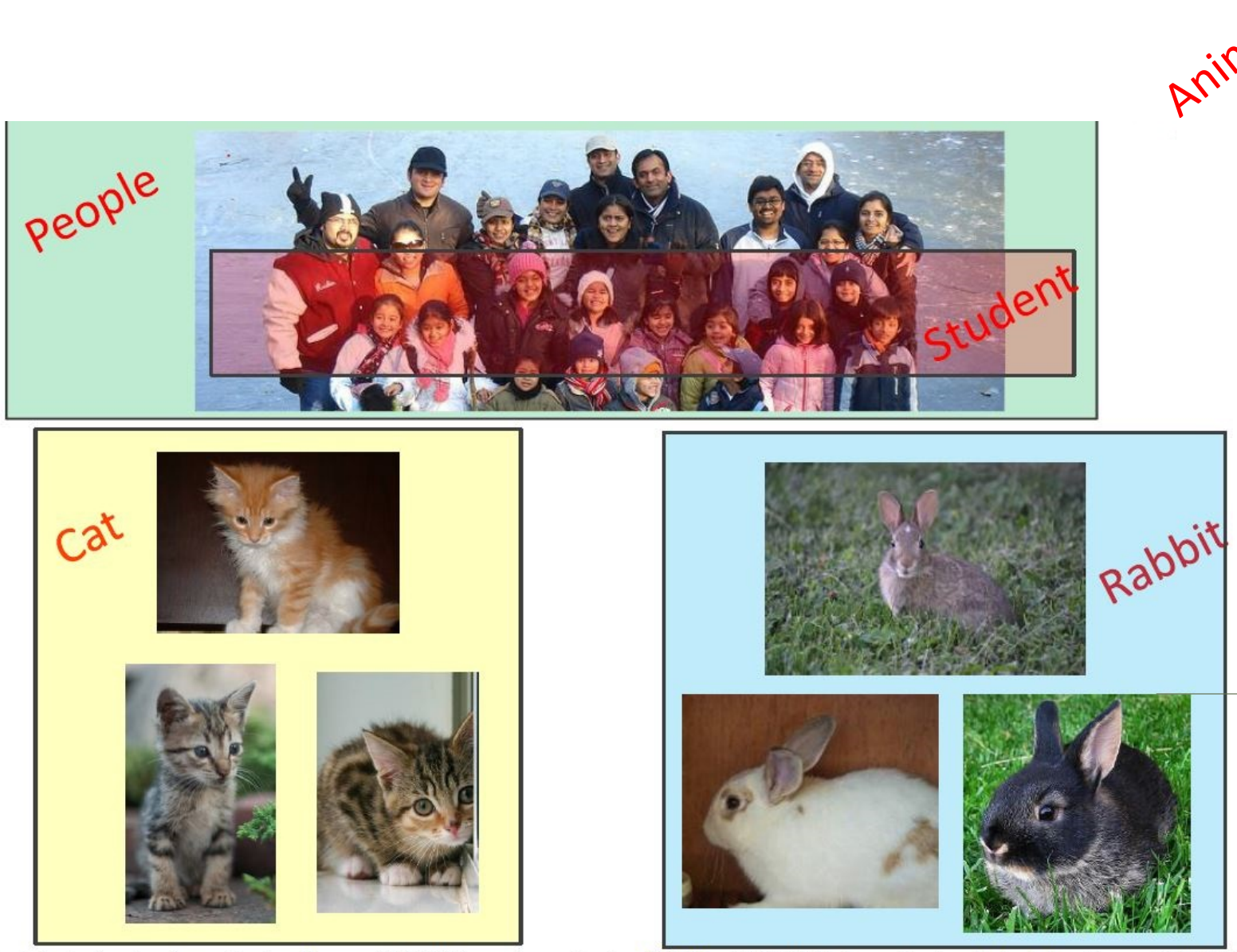
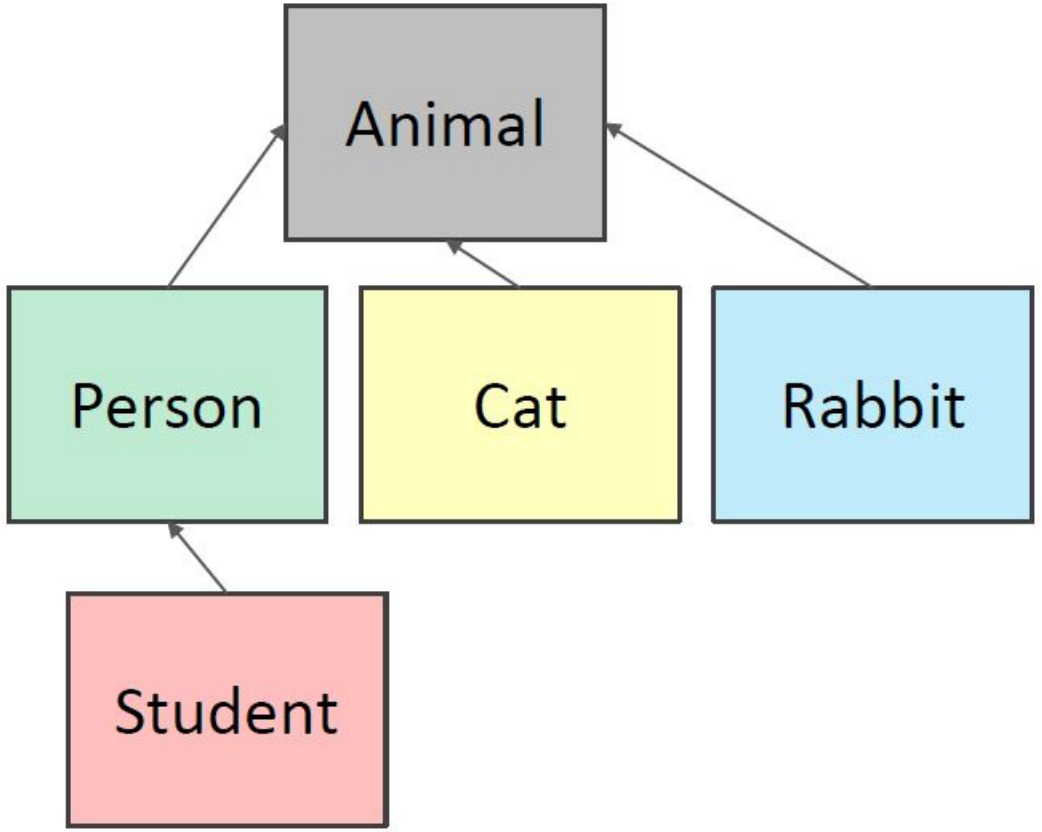



Image Credits, clockwise from top: Image Courtesy Deeeep, CC-BY-NC. Image Image Courtesy MTSOfan, CC-BY-NC-SA. Image Courtesy Carlos Solana, license CC-BY-NC-SA. Image Courtesy Rosemarie Banghart-Kovic, license CC-BY-NC-SA. Image Courtesy Paul Reynolds, license CC-BY. Image Courtesy Kenny Louie, License CC-BY. Courtesy Harald Wehner, in the public Domain.

# HIERARCHIES

- **parent class**(superclass)

- **child class**(subclass)
  - **inherits** all data and behaviors of parent class
  - **add** more **info**
  - **add** more **behavior**
  - **override** behavior

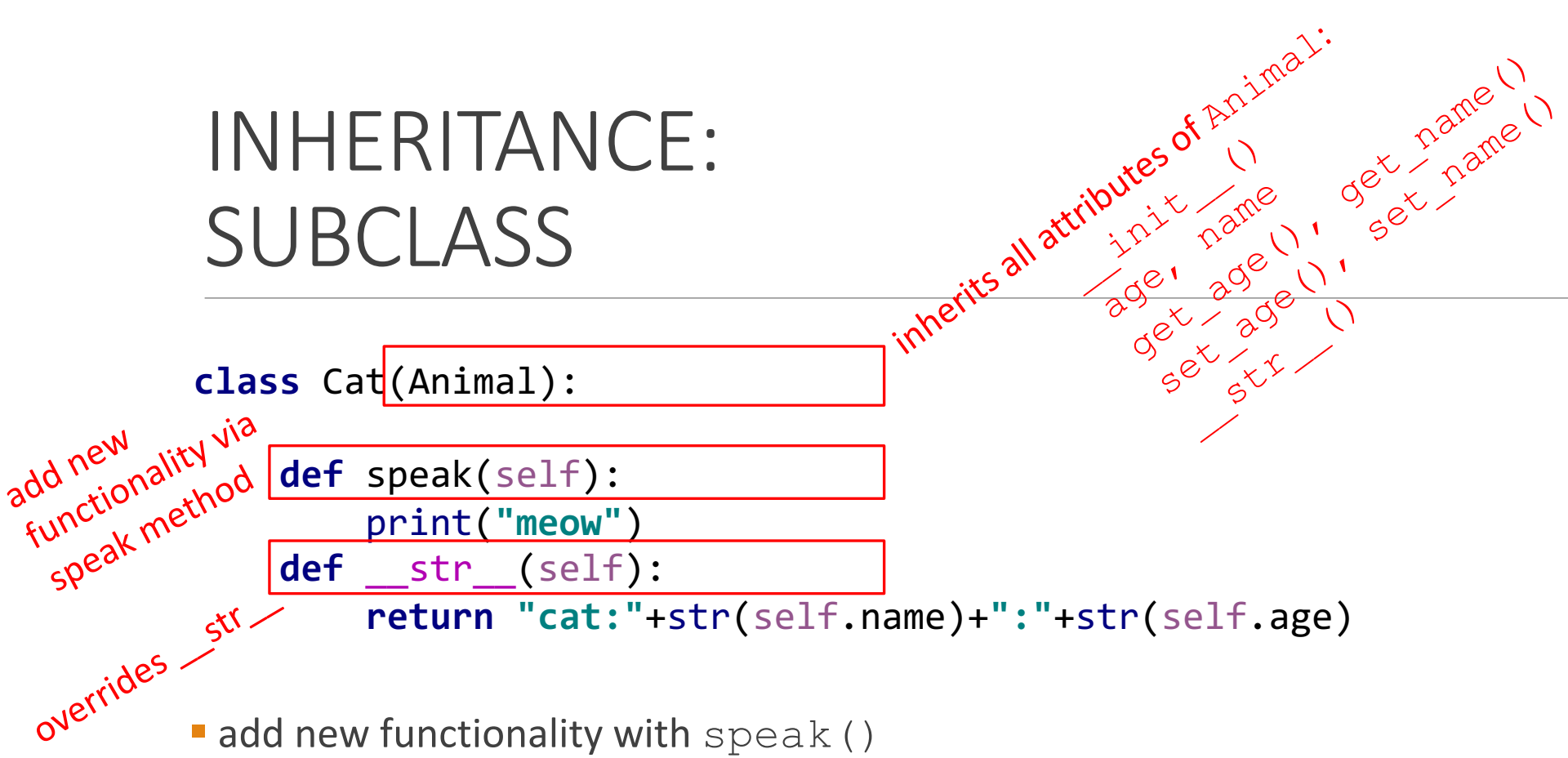


# INHERITANCE: PARENT CLASS

```
class Animal(object):
    def __init__(self, age):
        self.age = age
        self.name = None
    def get_age(self):
        return self.age
    def get_name(self):
        return self.name
    def set_age(self, newage):
        self.age = newage
    def set_name(self, newname=""):
        self.name = newname
    def __str__(self):
        return "animal:"+str(self.name)+":"+str(self.age)
```

- everything is an object
- class `object` implements basic operations in Python, like binding variables, etc
- you can avoid declaring `object` as parent class in Python 3+

# INHERITANCE: SUBCLASS




- add new functionality with `speak()`
  - instance of type `Cat` can be called with new methods
  - instance of type `Animal` throws error if called with Cat's new method
- `__init__` is not missing, uses the `Animal` version

# WHICH METHOD TO USE?

- subclass can have **methods with same name** as superclass
- for an instance of a class, look for a method name in **current class definition**
- if not found, look for method name **up the hierarchy** (in parent, then grandparent, and so on)
- use first method up the hierarchy that you found with that method name

```
class Person(Animal):
    def __init__(self, name, age):
        Animal.__init__(self, age)
        self.set_name(name)
        self.friends = []

    def get_friends(self):
        return self.friends
    def add_friend(self, fname):
        if fname not in self.friends:
            self.friends.append(fname)

    def speak(self):
        print("hello")

    def age_diff(self, other):
        diff = self.age-other.age
        print(abs(diff), "year difference")

    def __str__(self):
        return "person:"+str(self.name)+":"+str(self.age)
```

parent class is Animal

call Animal's constructor

call Animal's method

add new data attribute

new methods

override Animal's __str__ method

```python
import random

class Student(Person):
    def __init__(self, name, age, major=None):
        Person.__init__(self, name, age)
        self.major = major

    def change_major(self, major):
        self.major= major

    def speak(self):
        r = random.random()
        if r < 0.25:
            print("i have homework")
        elif 0.25 <= r < 0.5:
            print("i need sleep")
        elif 0.5 <= r < 0.75:
            print("i should eat")
        else:
            print("i am watching tv")

    def __str__(self):
        return "student:"+str(self.name)+":"+str(self.age)+":"+str(self.major)
```

bring in methods from `random` module

inherits Person and Animal attributes

adds new data

returns random float in [0, 1)

# OBJECT ORIENTED PROGRAMMING

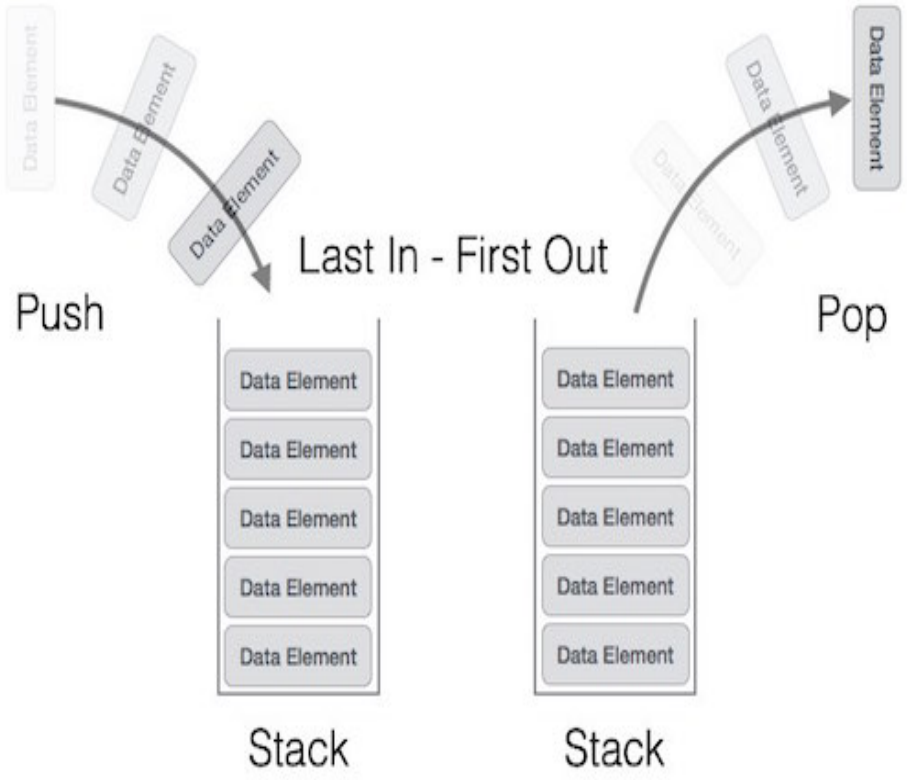
- create your own **collections of data**
- **organize** information
- **division** of work
- access information in a **consistent** manner
- add **layers** of complexity
- like functions, classes are a mechanism for **decomposition** and **abstraction** in programming

# PART II

# STACK

- a **collection** of items where the **last** added item is the **first** item that can be retrieved/removed

you can't pick a bottom element before removing the top elements


# STACK

- a linear sequence of data like list, and tuple
- support two main operations
  - **push**: which adds an element to the collection
  - **pop**: which removes the most recently added element that was not yet removed
- unlike lists,
  - items can be added only at one end
  - you only have access to top item
  - you can only remove the top item
- The order in which elements come off a stack gives rise to its alternative name, **LIFO (last in, first out).**

# STACK IMPLEMENTATION

- class name: `Stack`
- data attributes:
  - `container`: we use a list to store stack elements
  - `capacity`: an integer representing the max number of elements that a stack instance can have
- methods:
  - `__len__`: returns the size of stack
  - `is_full`: returns `True` iff the number of elements in stack equals to capacity
  - `is_empty`: returns `True` iff the stack is empty
  - `push`: adds a new element at the top of the stack
  - `pop`: removes the top element form stack
  - `top`: returns the top element without removing it
  - `__str__`: the string representation of a stack
- let's write the code for this. The file containing this code will be pushed to course repository for your reference.

# BUILT-IN STACK IN PYTHON?

- `list` in python has stack interface which enables you to treat a list as a stack!

initializing an empty stack

```
s = []
```

```
s.append(1)
s.append(2)
```

equivalent to **push**

```
print(s[-1])
```

equivalent to **top**

```
s.pop()
```

like stack **pop**

# STACK IN ACTION

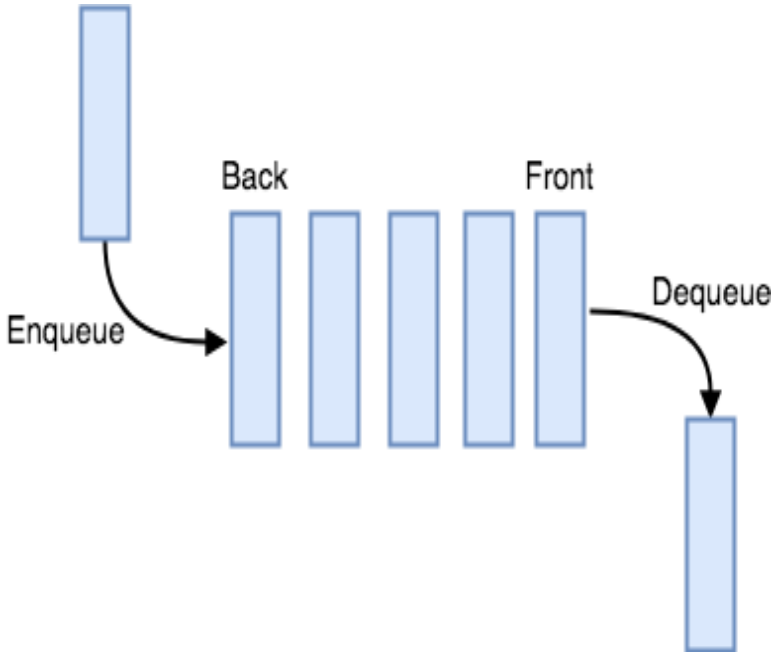
- you are given a string like ‘abcd’, by using a stack try to get the reverse of the given string.

# STACK and RECURSION

- A stack is a **recursive** data structure. Here is a structural definition of a Stack:
  - a stack is either ***empty*** or
  - it consists of ***a top and the rest which is a stack***
- recursion is closely tied to stack
  - the recursive function calls are stored in a stack
  - they will return on the LIFO principle
- you can devise iterative solutions for recursive problems using an explicit stack!

# QUEUE

- a **collection** of items where the **first** added item is the **first** item that can be retrieved/removed




# QUEUE

- a linear sequence of data like list, and tuple, stack
- support two main operations
  - **enqueue**: which adds an element to the collection
  - **dequeue**: which removes the first added element that was not yet removed
- unlike lists,
  - items can be added only at one end
  - you only have access to the front item (also back item if you have a double ended queue)
  - you can only remove the item at one end
- The order in which elements come off a queue gives rise to its alternative name, **FIFO (first in, first out).**

# QUEUE IMPLEMENTATION

- class name: `Queue`
- data attributes:
  - `container`: we use a list to store stack elements

we assume unbounded capacity

- methods:
  - `__len__`: returns the size of queue
  - `is_empty`: returns `True` iff the queue is empty
  - `enqueue`: adds a new element at the back of the queue
  - `dequeue`: removes and returns the element at the front of the queue
  - `front`: returns the element at the front without removing
  - `__str__`: the string representation of the queue
- let's write the code for this. The file containing this code will be pushed to course repository for your reference.

# BUILT-IN QUEUE IN PYTHON?

- `queue` module in python has a number of queue implementations.

```
from queue import Queue
```

importing `Queue` class from `queue` module

initializing an empty queue

```
q = Queue()
```

```
q.put('a')
q.put('b')
```

equivalent to **enqueue**

equivalent to **is_empty**

```
print(q.empty())
```

```
print(q.get())
```

equivalent to **dequeue**

`Queue` does not have an equivalent method to `front()`

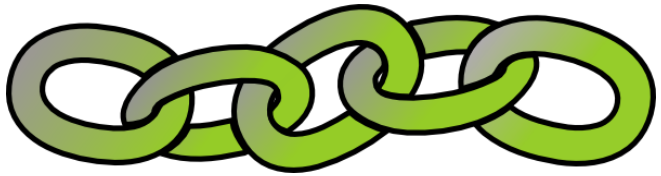
# QUEUE APPLICATIONS

- used when things don't have to be processed immediately, but have to be processed in First In First Out order like Breadth First Search.

we will learn this later

- When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
- When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples include IO Buffers, pipes, file IO, etc.

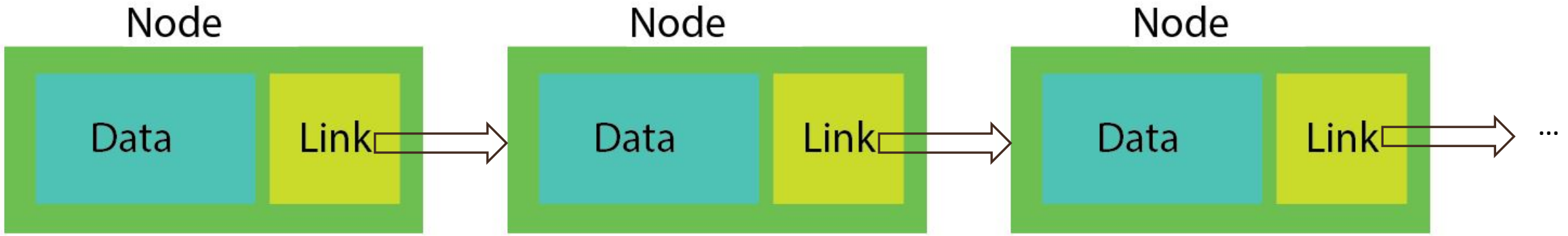
# LINKED LIST



- a linear collection of data elements, whose order is not given by their physical placement in memory. Instead, each element **points** to the next. It is a data structure consisting of a collection of **nodes** which together represent a sequence.
- we have multiple variations of Linked Lists, like **singly linked lists** and doubly linked lists. We only introduce singly linked list here.

# LINKED LIST is a CHAIN of NODES

- each node contains: **data**, and a **reference** (in other words, a link) to the next node in the sequence



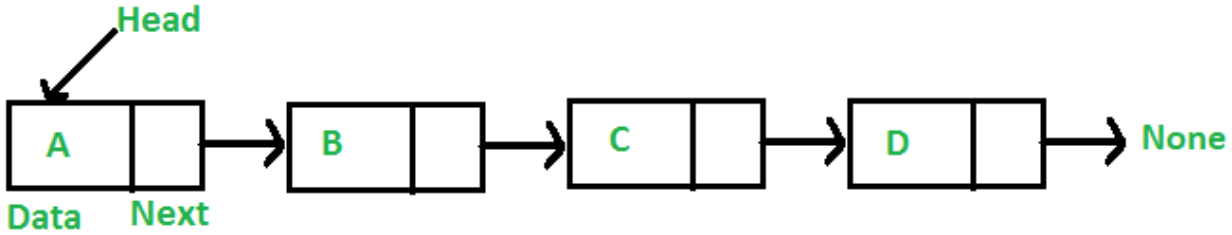
# NODE CLASS

```
class Node:

    def __init__(self, data):
        self.data = data
        self.next = None   # next Node after self in a LinkedList that self is
```

# LINKED LIST

- a linear sequence of data like list, and tuple, stack, queue
- like lists
  - items can be added anywhere in the sequence
  - you can access any element in the sequence
  - you can remove any elements
- unlike lists
  - no constant access to elements (you need to start from head and follow links to reach your desired element)
  - removals and insertions takes places in constant time (in list, you may need to shift elements)
- Linked lists are dynamic, so the length of list can increase or decrease as necessary. Each node does not necessarily follow the previous one physically in the memory.

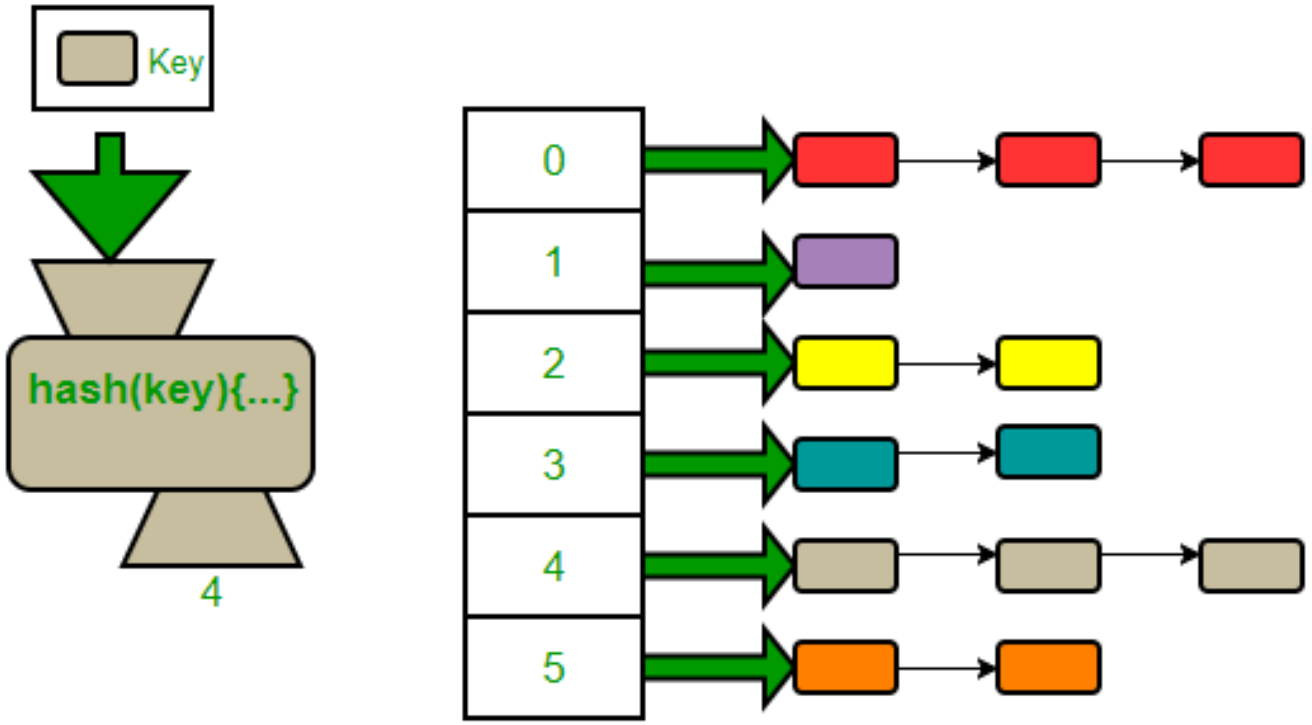


# LINKED LIST IMPLEMENTATION

- class name: `LinkedList`
- data attributes:
  - `head`: the starting node in the linked list
  - `size`: the number of nodes in the linked list
- methods:
  - `__len__`: returns the size of linked list
  - `is_empty`: returns `True` iff the linked list is empty
  - `append`: insert a given node at the end of linked list
  - `find`: return a node which has the given data
  - `remove`: removes a given node in the linked list
  - `__str__`: the string representation of the linked list
- let's write the code for this. The file containing this code will be pushed to course repository for your reference.

# BUILT-IN LINKED LIST in PYTHON

- As of Python 3.7, doesn't provide a dedicated linked list data type. There's nothing like Java's LinkedList built into Python or into the Python standard library.
- Python does however include the `collections.deque` class which provides a double-ended queue and is implemented as a doubly-linked list internally.
  Under some specific circumstances you might be able to use it as a "makeshift" linked list. If that's not an option you'll need to write your own linked list implementation from scratch.

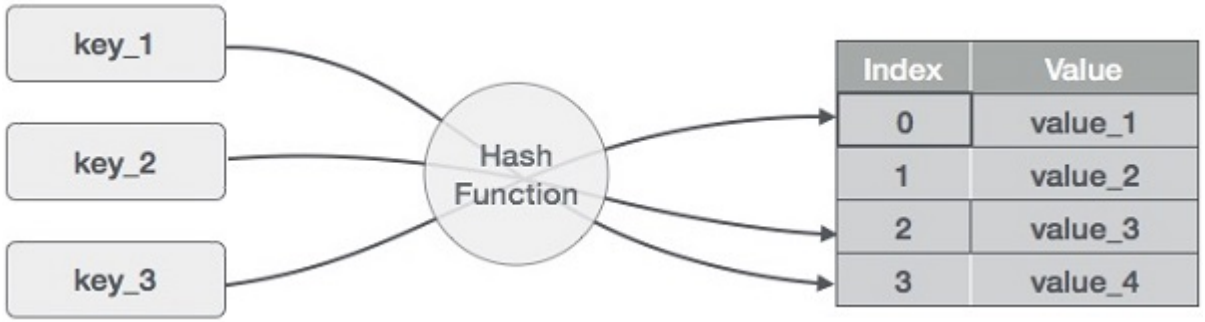



# HASH TABLE

- a structure that can **map keys to values**. A hash table uses a **hash function** to compute an index, also called a hash code, into an array of buckets or slots, from which the desired value can be found.

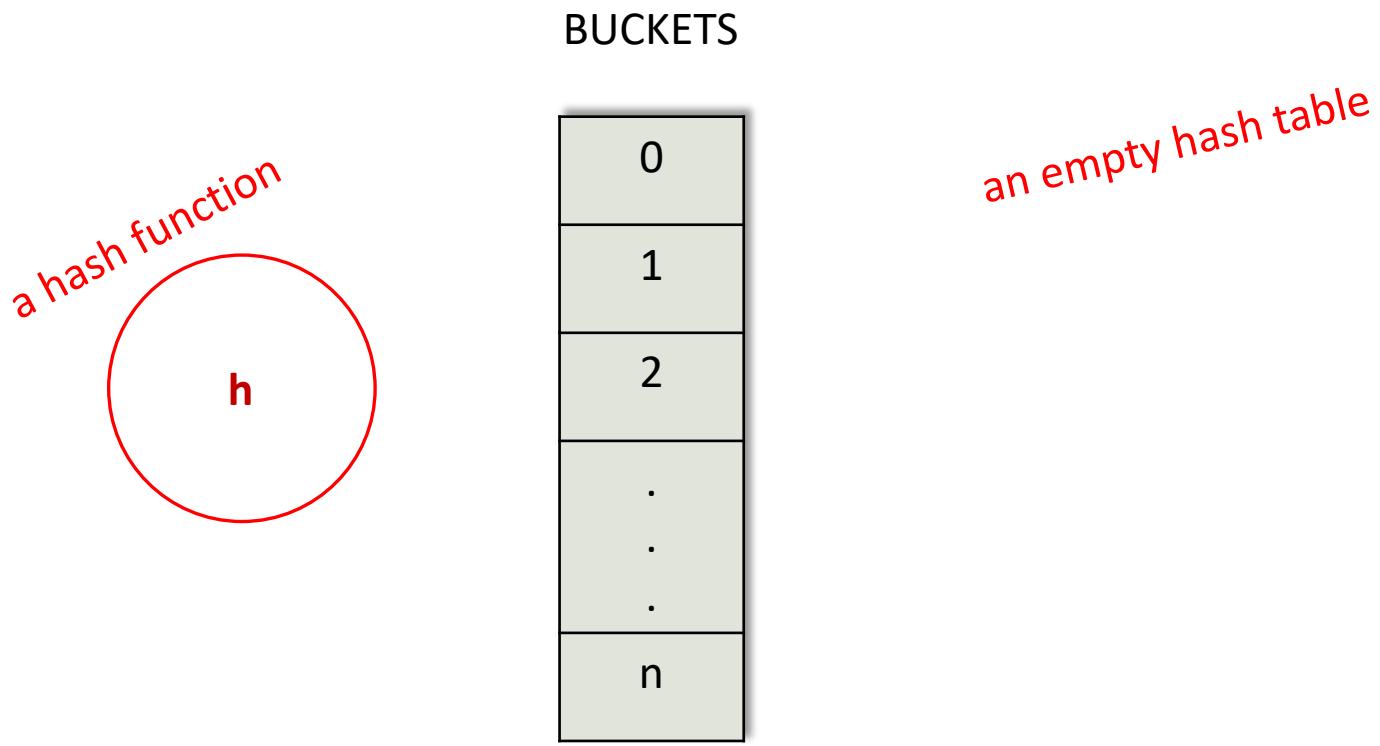
# HASH TABLE

- a collection of **key-value** pairs of data
- support two main operations
  - **put**: adds a given key-value pair to the collection
  - **get**: returns the value associated with a given key
- unlike other sequences we've seen so far
  - items are not stored as a linear sequence
- adding/retrieving a pair to/from a Hash Table is expected to happen in **constant time**
- the elements that come off a Hash Table do not necessarily follow any specific order

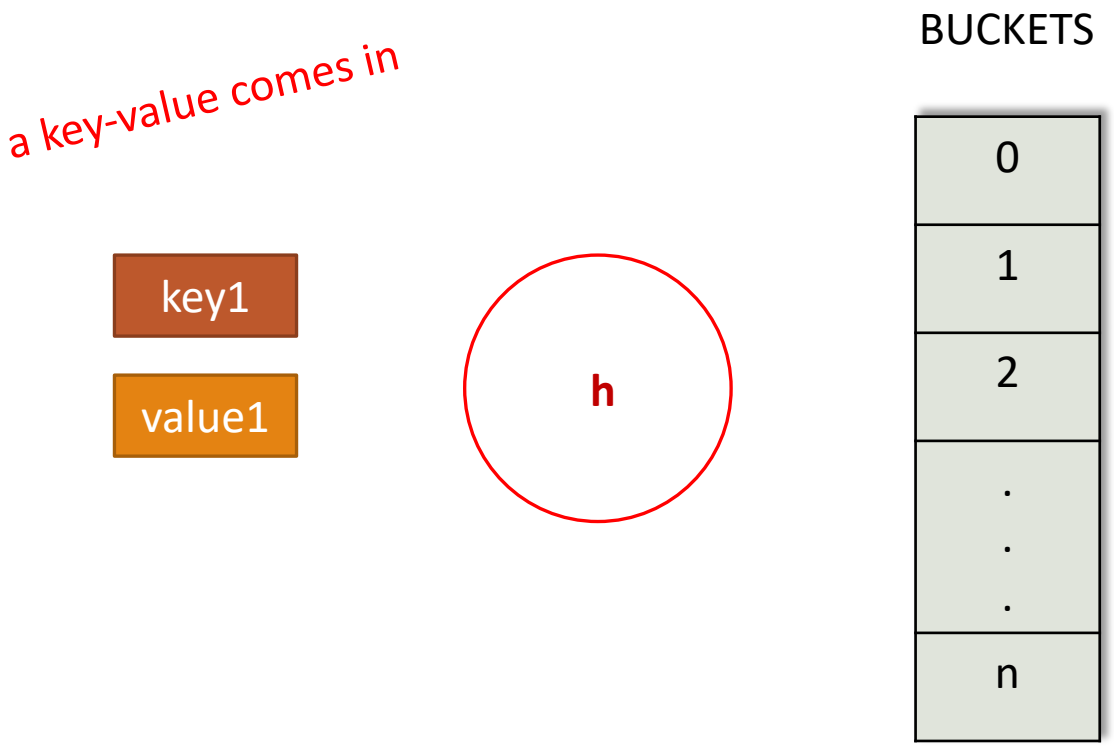



will talk more about this later

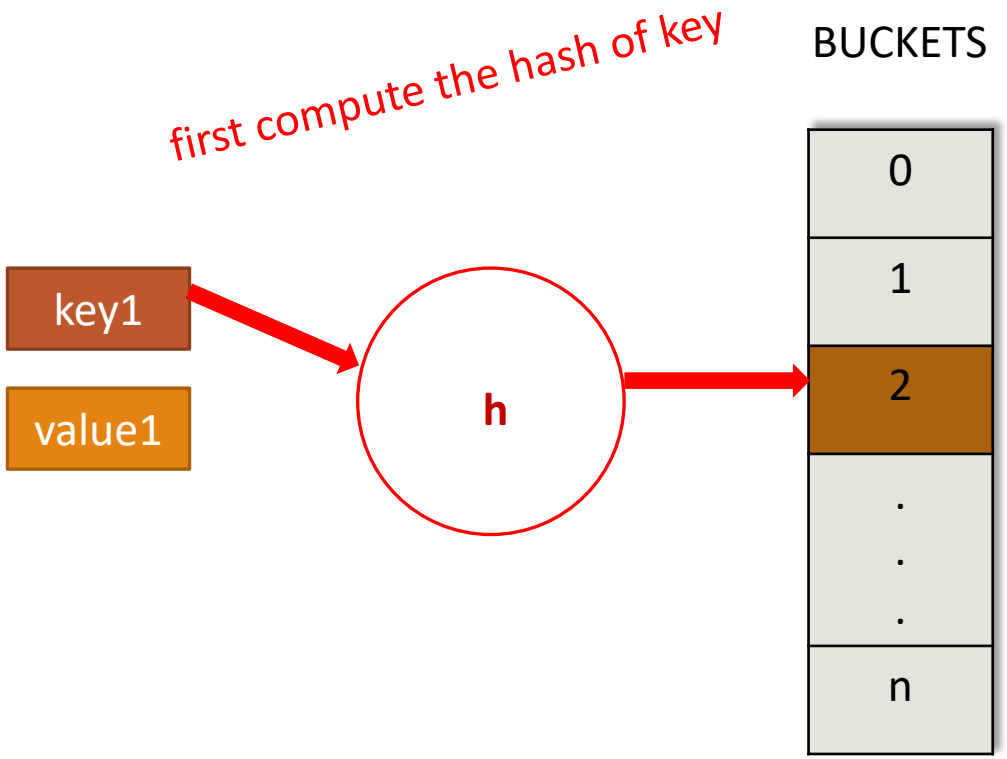
# INTERNAL STRUCTURE



# INTERNAL STRUCTURE

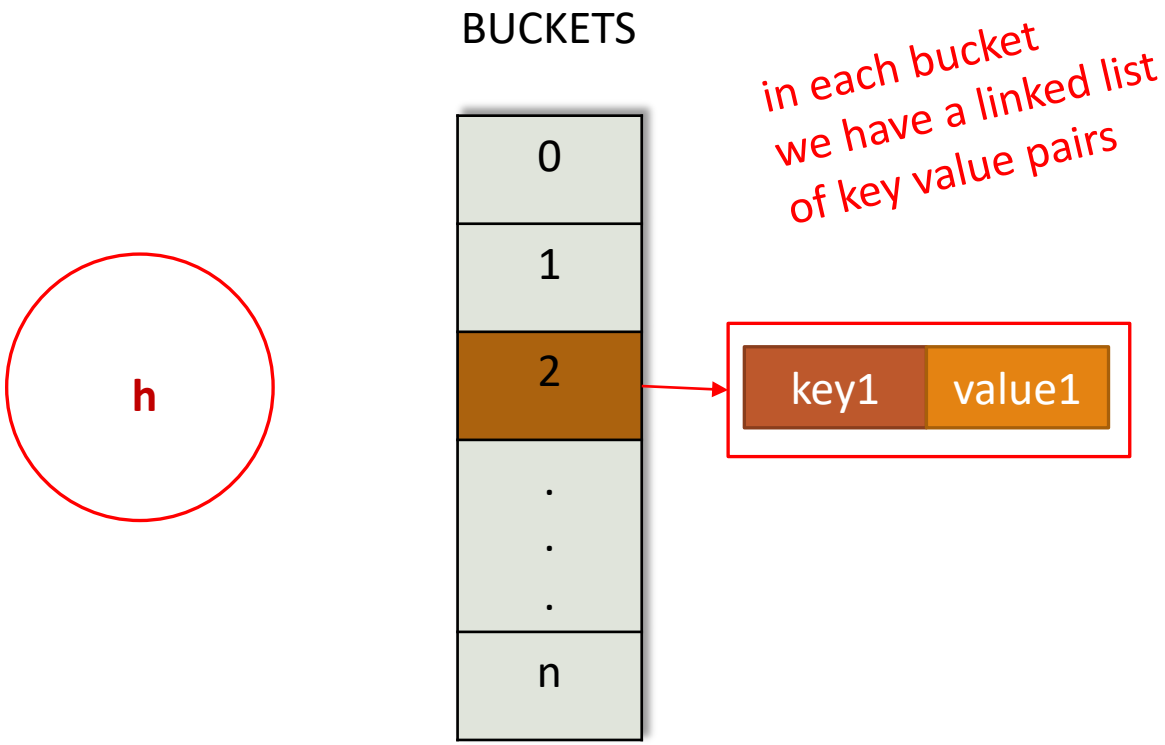


# INTERNAL STRUCTURE

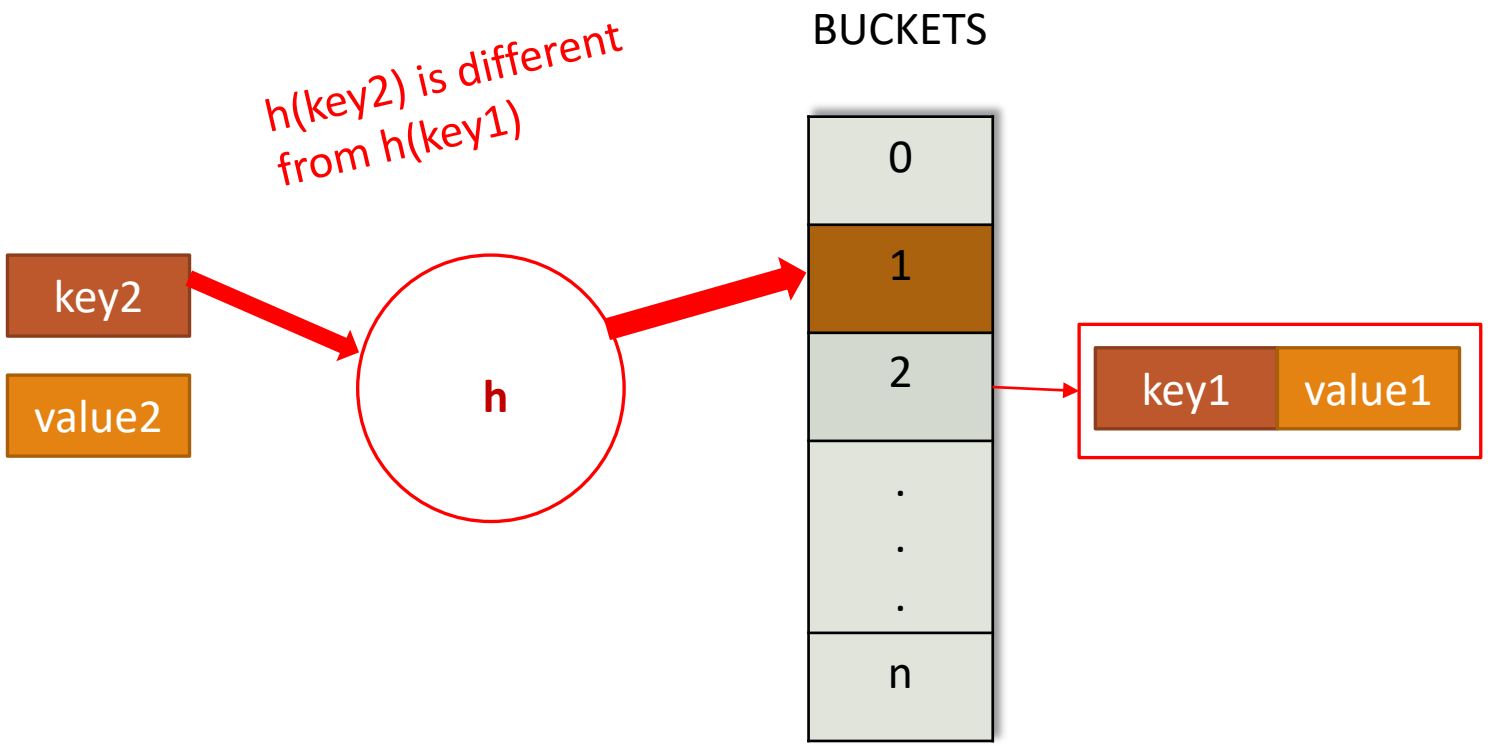



- **hash function** accepts keys as input, and outputs an **integer** which represents a bucket **index**
- the key-value pair like `(k,v)` will be put into a bucket, with `index = h(k)`

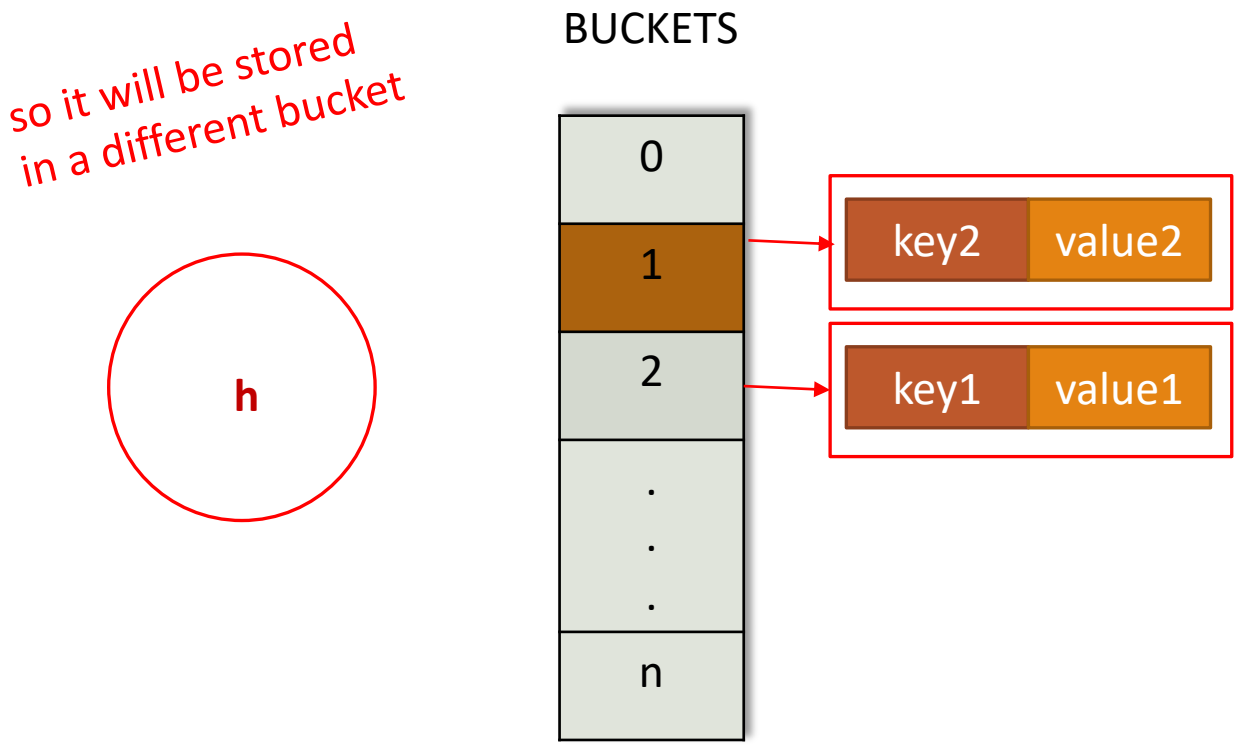
# INTERNAL STRUCTURE



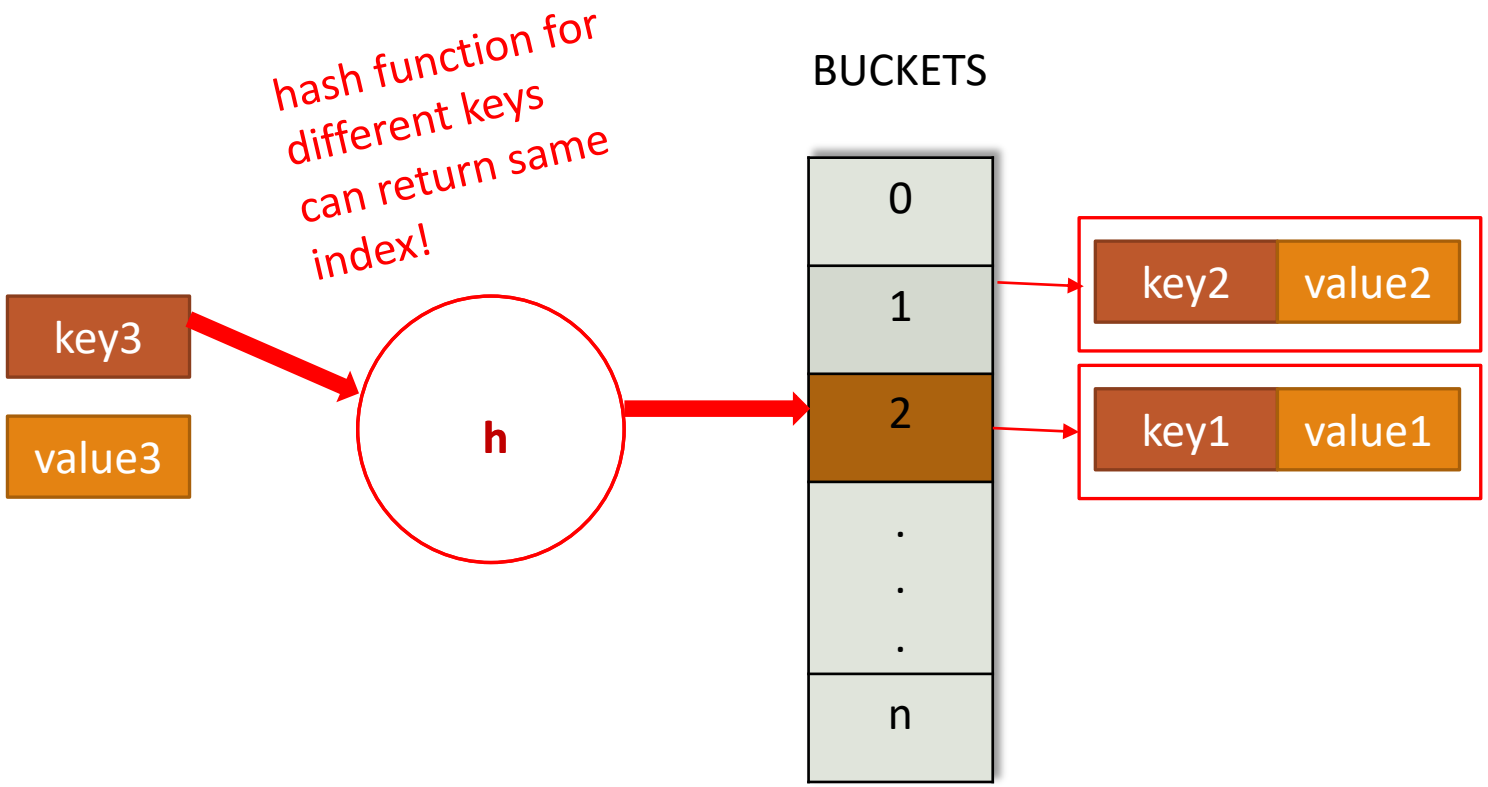
# INTERNAL STRUCTURE



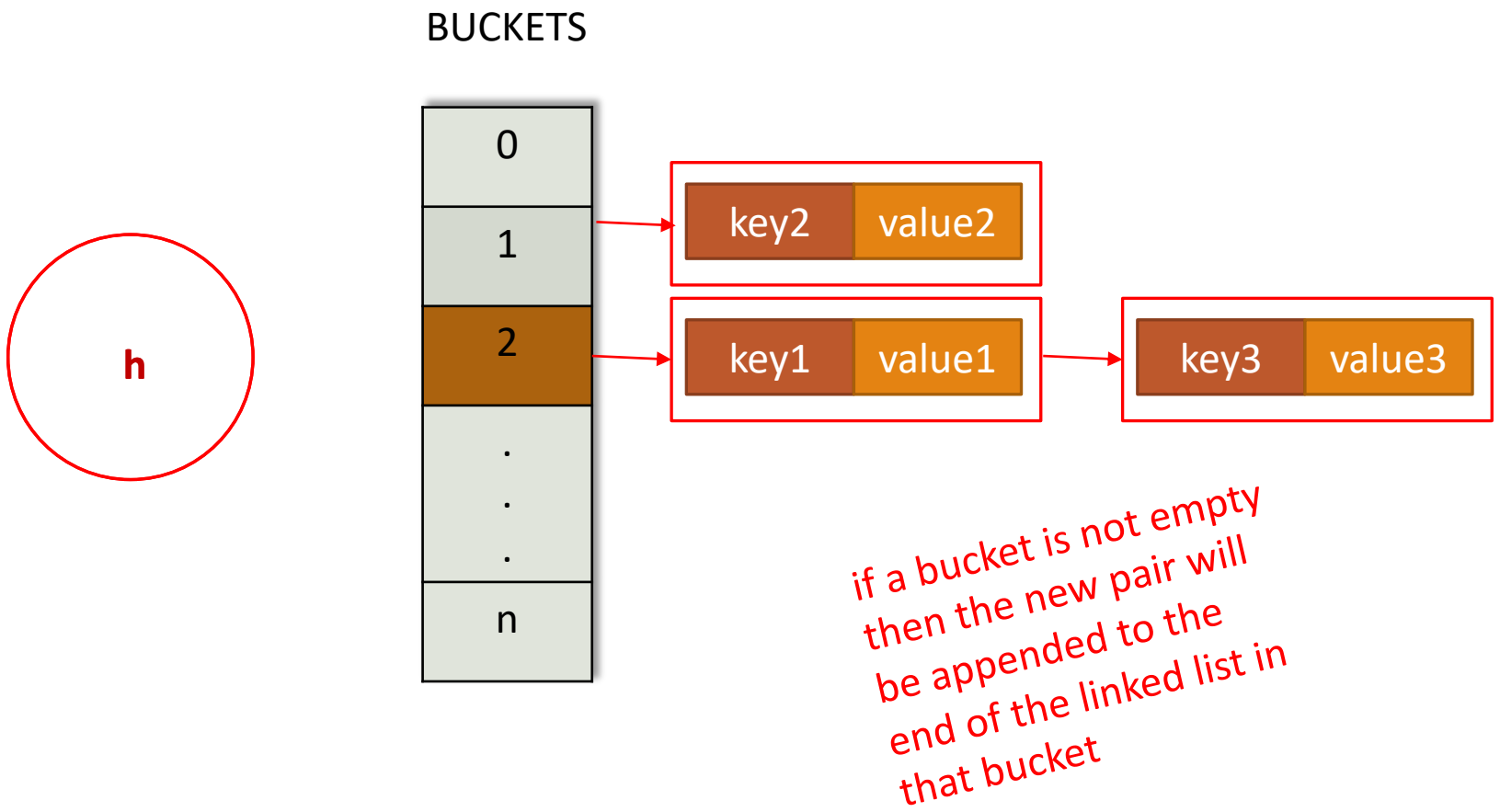
# INTERNAL STRUCTURE



# INTERNAL STRUCTURE

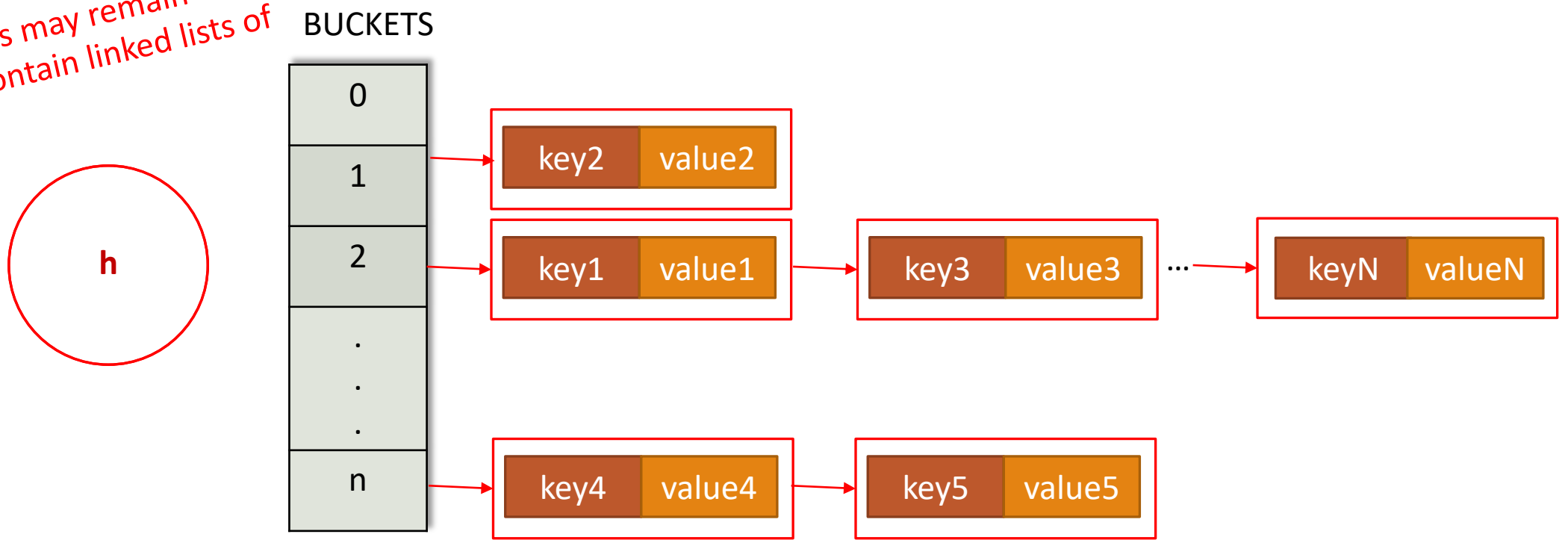


# INTERNAL STRUCTURE

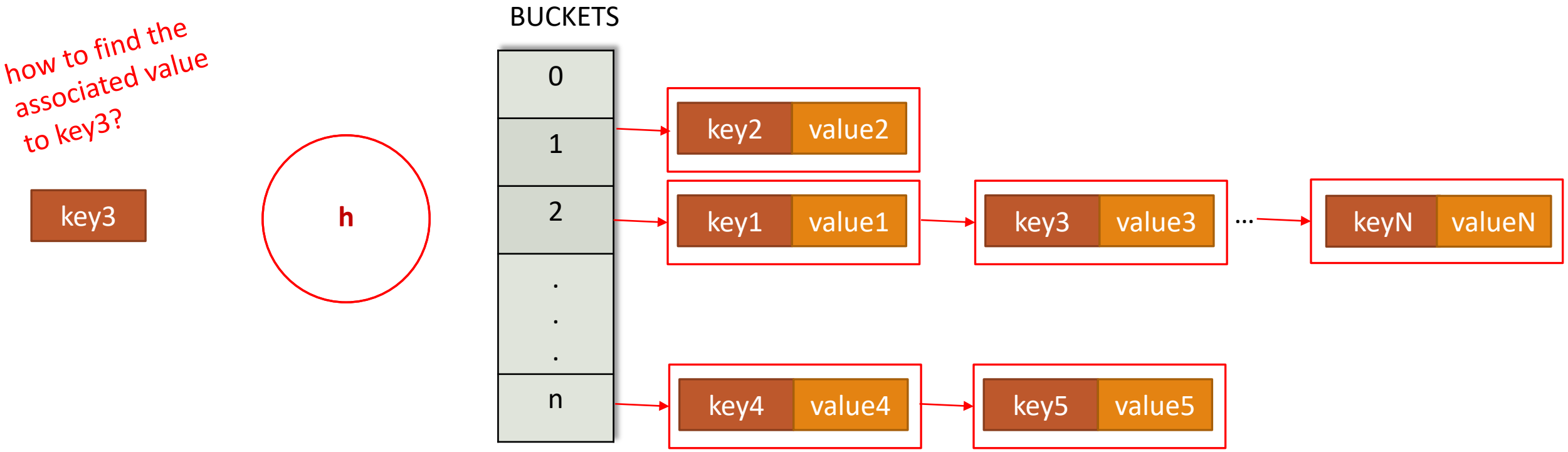


# INTERNAL STRUCTURE

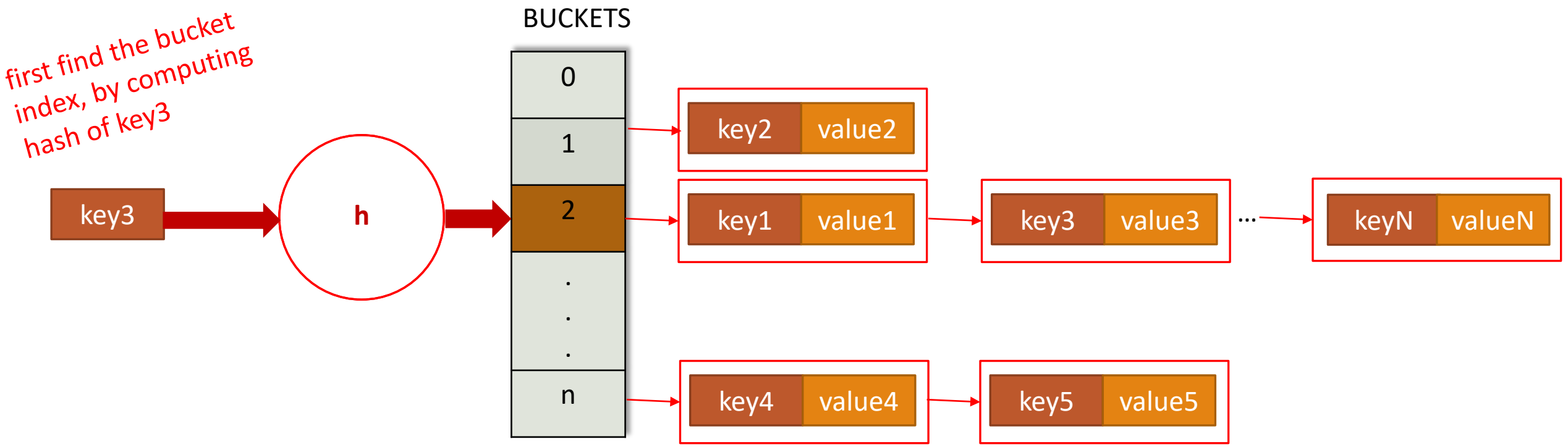
finally, some buckets may remain empty and some others contain linked lists of key-value pairs



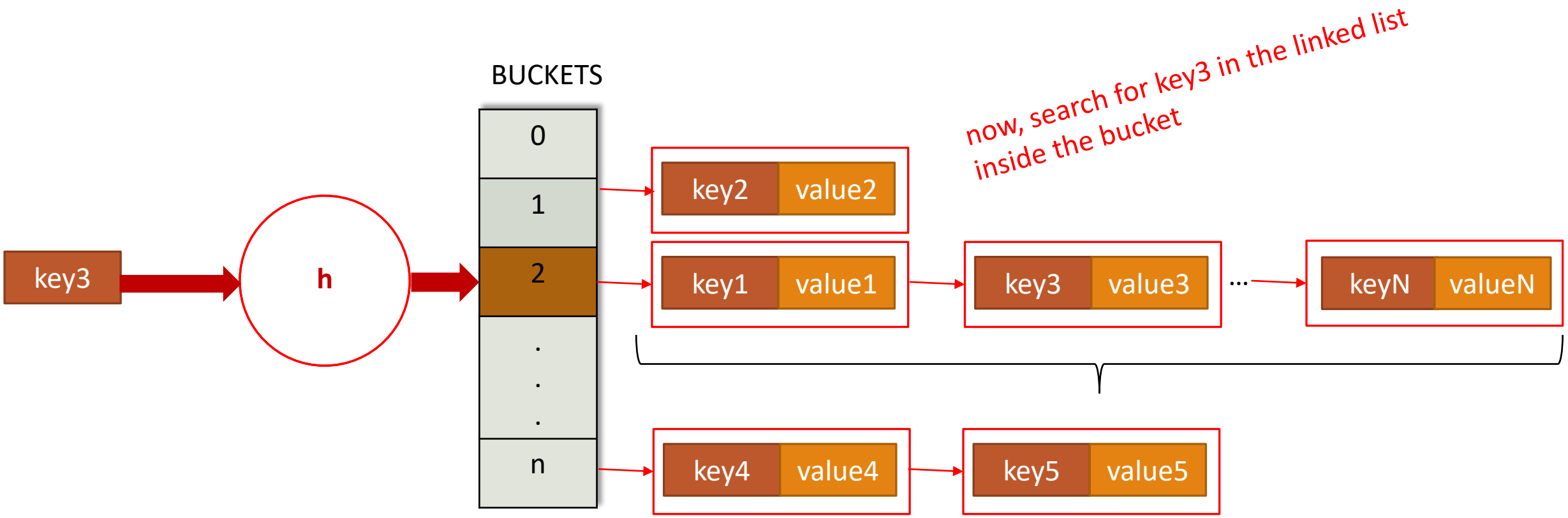
# RETRIEVING VALUE for a KEY



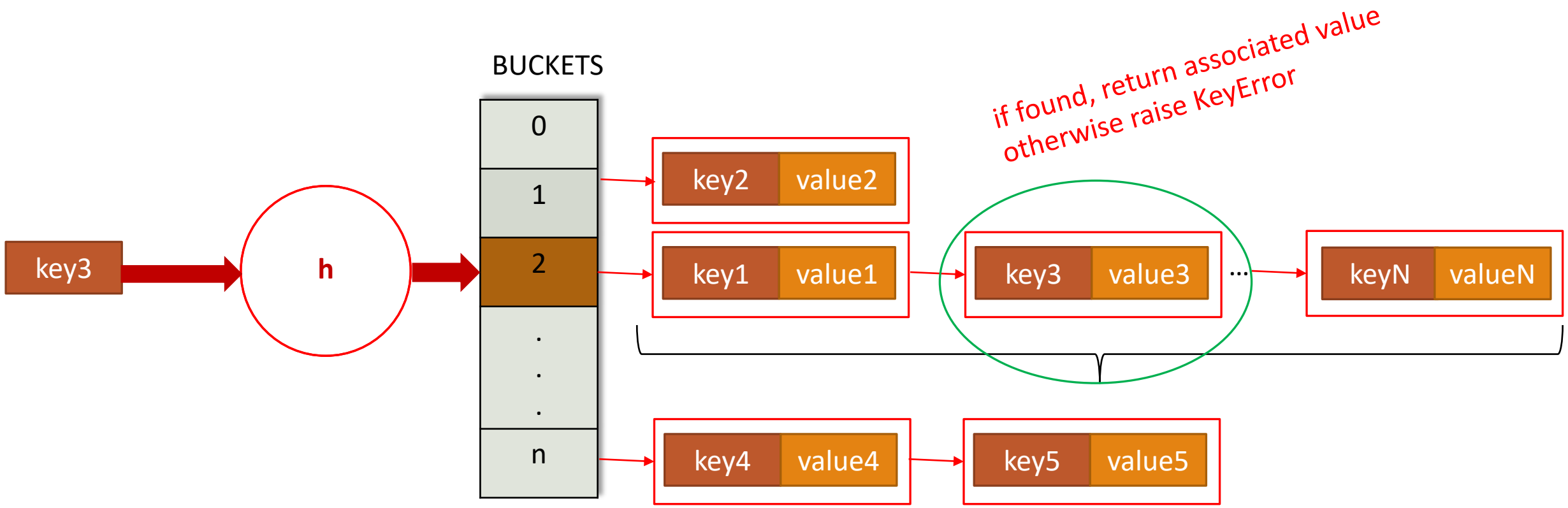
# RETRIEVING VALUE for a KEY



# RETRIEVING VALUE for a KEY



# RETRIEVING VALUE for a KEY



# HASH FUNCTION

- the idea of hashing is to distribute the entries (key/value pairs) across an array of **buckets**
- a good hash function is a function which results in less **collisions**
- a critical statistic for a hash table is the *load factor*, defined as

$$load\ factor = \frac{n}{k}$$

where

- *n* is the number of entries occupied in the hash table.
- *k* is the number of buckets.

- as the load factor grows larger, the hash table becomes slower, and it may even fail to work
- the expected constant time property of a hash table assumes that the load factor be kept below some bound

# BUILT-IN HASH TABLE in PYTHON?

- dictionaries!