Data Structure

Omar Abdul-Latif

Data Structure

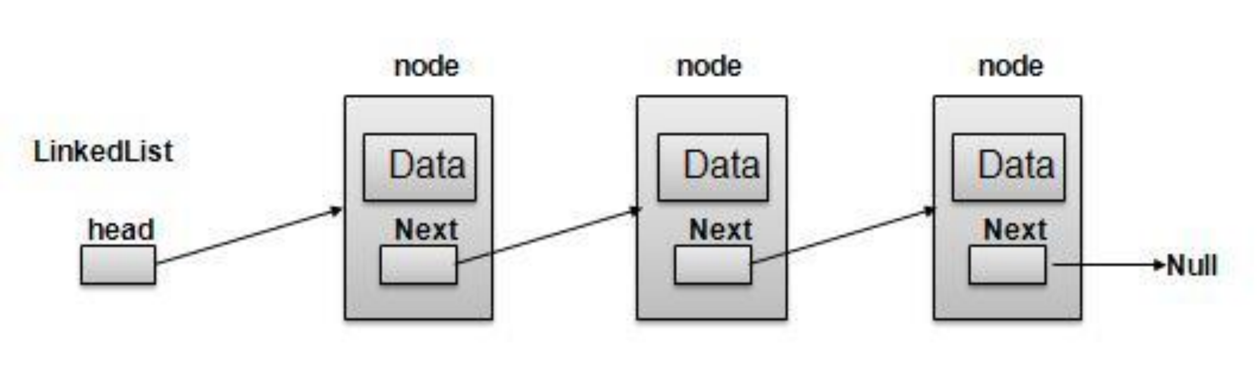
- What is Data Structure?
  - A data structure is a collection of data organized in some fashion. A data structure not only stores data, but also supports the operations for manipulating data in the structure.
  - In object-oriented thinking, a data structure is an object that stores other objects, referred to as data or elements.
  - So sometimes it is referred to as a ***container*** *object* or a ***collection*** *object*.

The Four Classic Data Structures

- Linked Lists
- Stacks
- Queues
- Priority Queues

Linked Lists

- A linked-list is a sequence of data structures which are connected together via links.
- Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list the second most used data structure after array.

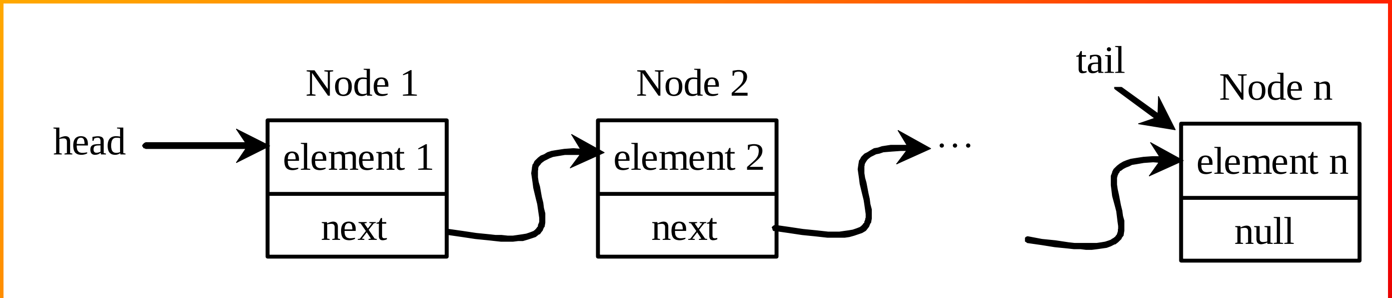


Types of Linked Lists

- Singly Linked Lists (basic/default)
  - Ordered Linked Lists
  - Unordered Linked Lists
- Circular Linked Lists
- Doubly Linked Lists
- Circular Doubly Linked Lists

Nodes in Singly Linked Lists

A linked list consists of nodes. Each node contains an element, and each node is linked to its next neighbor. Thus a node can be defined as a class, as follows:

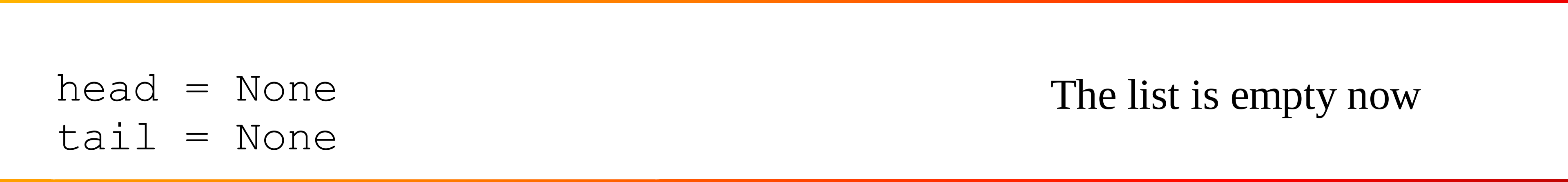


```
class Node:
    def __init__(self, element):
        self.elmenet = element
        self.next = None
```

Adding Three Nodes

The variable head refers to the first node in the list, and the variable tail refers to the last node in the list. If the list is empty, both are None. For example, you can create three nodes to store three strings in a list, as follows:

Step 1: Declare head and tail:



```
head = None
tail = None
```

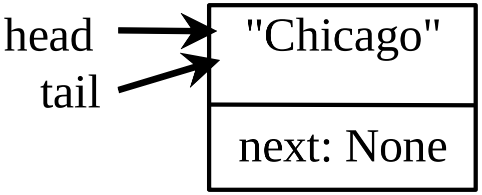
The list is empty now

Adding Three Nodes, cont.

Step 2: Create the first node and insert it to the list:

```
head = Node("Chicago")
tail = head
```

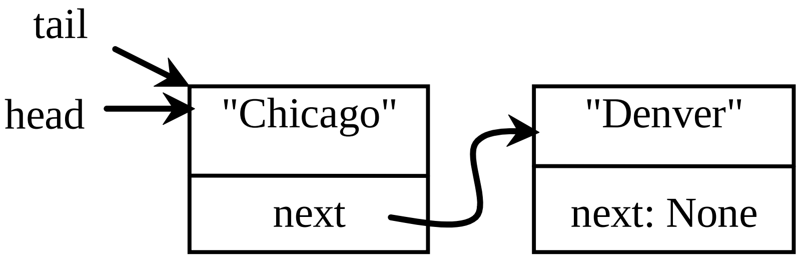
After the first node is inserted



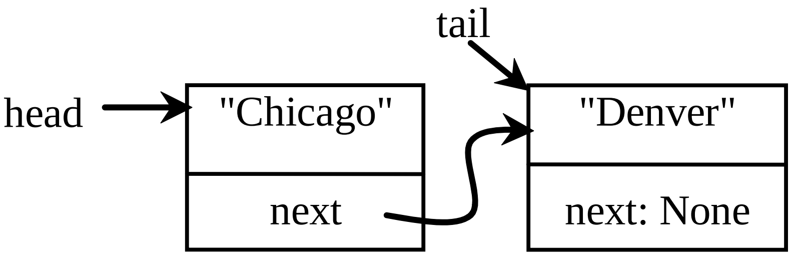
Adding Three Nodes, cont.

Step 3: Create the second node and insert it to the list:

```
tail.next = Node("Denver")
```



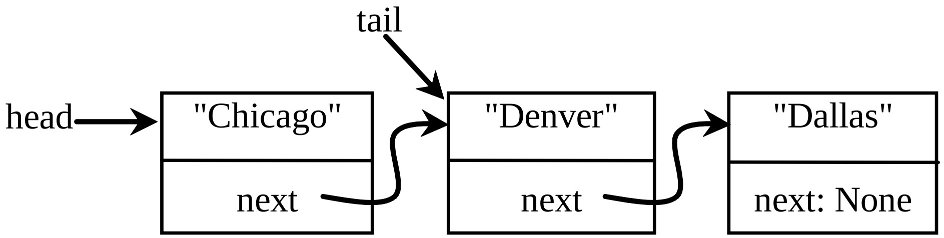
```
tail  = tail.next
```



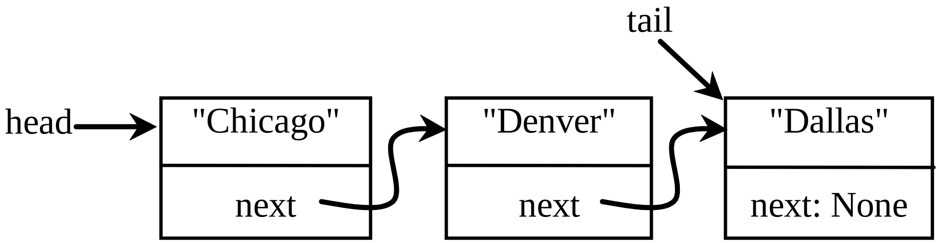
Adding Three Nodes, cont.

Step 4: Create the third node and insert it to the list:

```
tail.next = Node("Dallas")
```



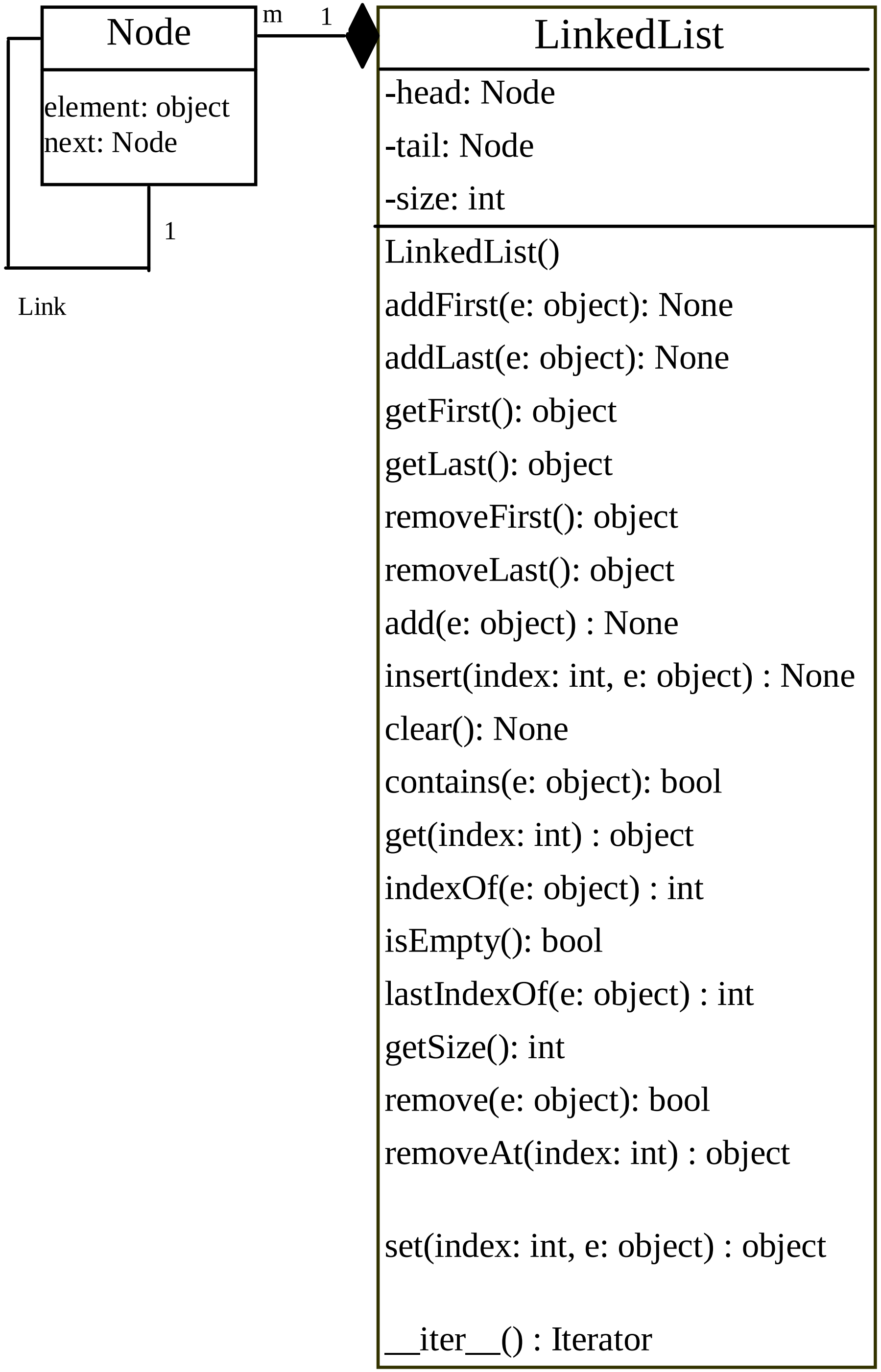
```
tail = tail.next
```



Traversing All Elements in the List

Each node contains the element and a data field named *next* that points to the next element. If the node is the last in the list, its pointer data field next contains the value None. You can use this property to detect the last node. For example, you may write the following loop to traverse all the nodes in the list.

```
current = head
while current != None:
    print(current.element)
    current = current.next
```



Node

element: object
next: Node

LinkedList

-head: Node

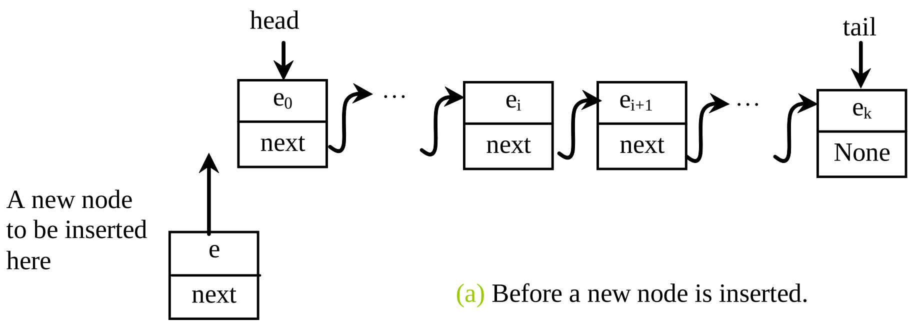
-tail: Node

-size: int

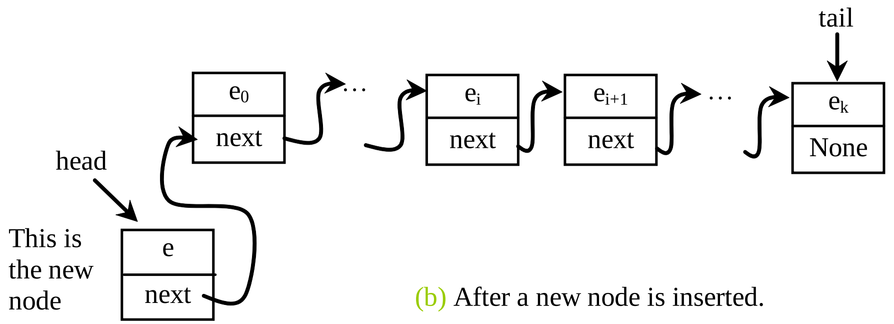
LinkedList()	Creates an empty linked list.
addFirst(e: object): None	Adds a new element to the head of the list.
addLast(e: object): None	Adds a new element to the tail of the list.
getFirst(): object	Returns the first element in the list.
getLast(): object	Returns the last element in the list.
removeFirst(): object	Removes the first element from the list.
removeLast(): object	Removes the last element from the list.
add(e: object) : None	Same as addLast(e).
insert(index: int, e: object) : None	Adds a new element at the specified index.
clear(): None	Removes all the elements from this list.
contains(e: object): bool	Returns true if this list contains the element.
get(index: int) : object	Returns the element from at the specified index.
indexOf(e: object) : int	Returns the index of the first matching element.
isEmpty(): bool	Returns true if this list contains no elements.
lastIndexOf(e: object) : int	Returns the index of the last matching element.
getSize(): int	Returns the number of elements in this list.
remove(e: object): bool	Removes the element from this list.
removeAt(index: int) : object	Removes the element at the specified index and returns the removed element.
set(index: int, e: object) : object	Sets the element at the specified index and returns the element you are replacing.
__iter__() : Iterator	Returns an iterator for this linked list.

Implementing addFirst(e)

```
def addFirst(self, e):
    newNode = Node(e) # Create a new node
    newNode.next = self.__head # link the new node with the head
    self.__head = newNode # head points to the new node
    self.__size += 1 # Increase list size
    if self.__tail == None: # the new node is the only node in list
        self.__tail = self.__head
```



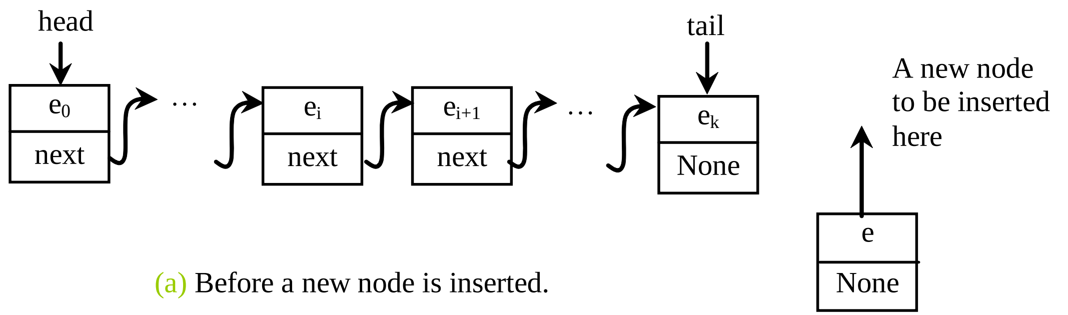
(a) Before a new node is inserted.



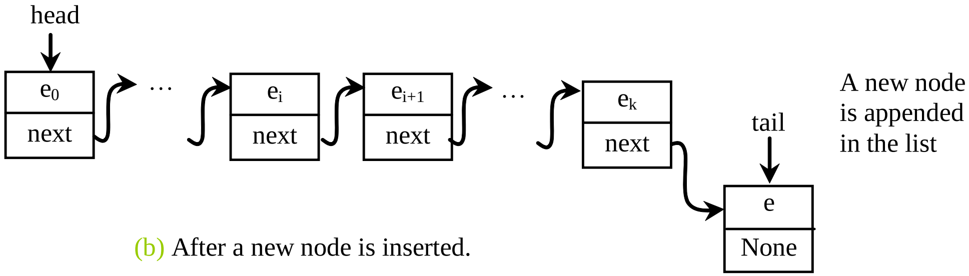
(b) After a new node is inserted.

Implementing addLast(e)

```
def addLast(self, e):
   newNode = Node(e) # Create a new node for e

   if self.__tail == None:
      self.__head = self.__tail = newNode # The only node in list
   else:
      self.__tail.next = newNode # Link the new with the last node
      self.__tail = self.__tail.next # tail now points to the last node

   self.__size += 1 # Increase size
```



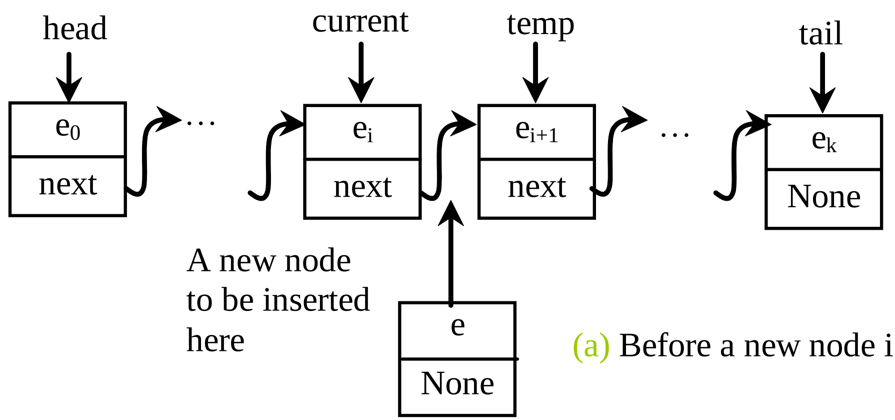
(a) Before a new node is inserted.



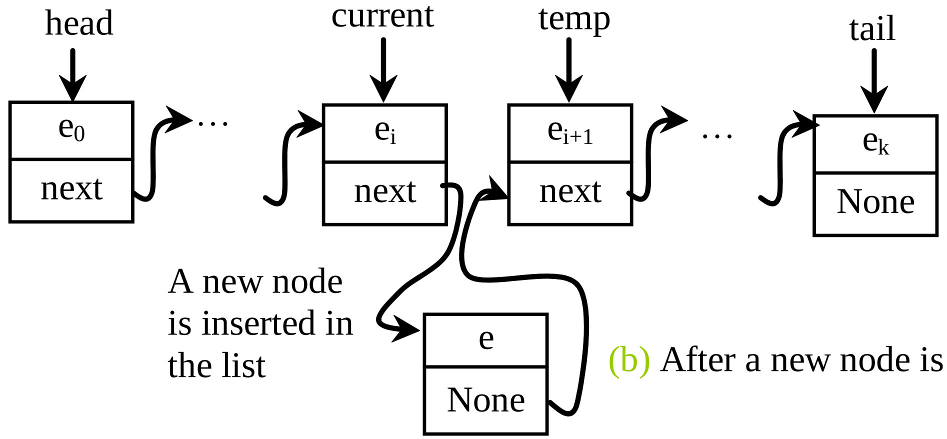
(b) After a new node is inserted.

Implementing add(index, e)

```
def insert(self, index, e):
    if index == 0:
        self.addFirst(e) # Insert first
    elif index >= size:
        self.addLast(e) # Insert last
    else: # Insert in the middle
        current = head
        for i in range(1, index):
            current = current.next
        temp = current.next
        current.next = Node(e)
        (current.next).next = temp
        self.__size += 1
```



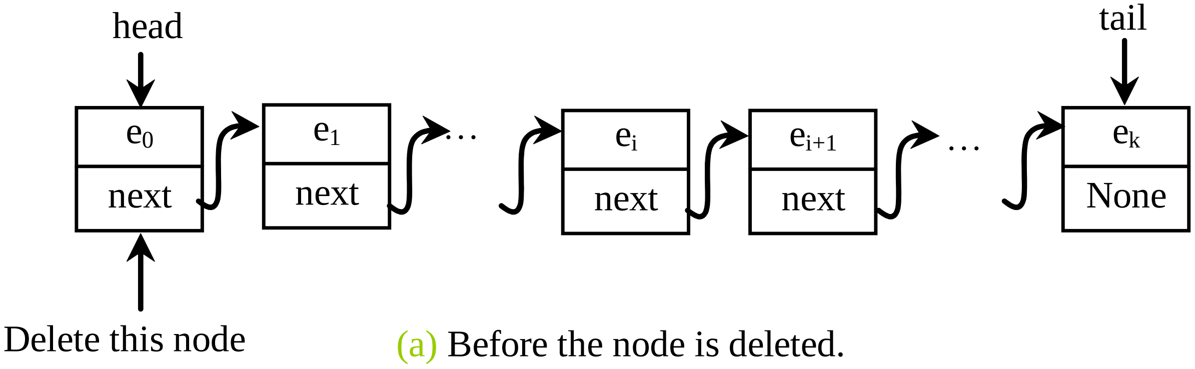
(a) Before a new node is inserted.



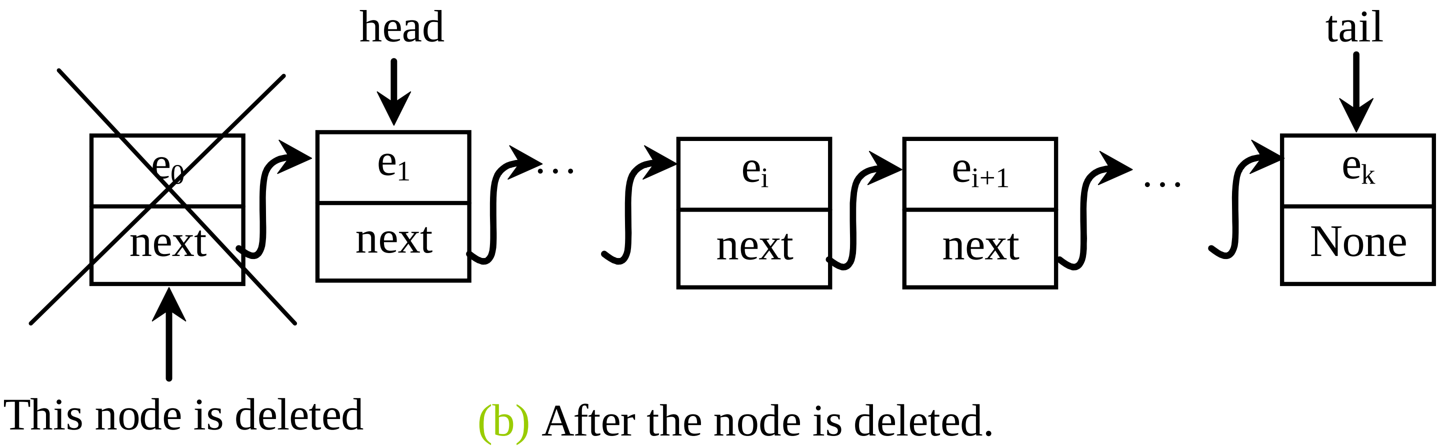
(b) After a new node is inserted.

Implementing removeFirst()

```
def removeFirst(self):
    if self.__size == 0:
        return None
    else:
        temp = self.__head
        self.__head = self.__head.next
        self.__size -= 1
    if self.__head == None:
       self.__tail = None
    return temp.element
```



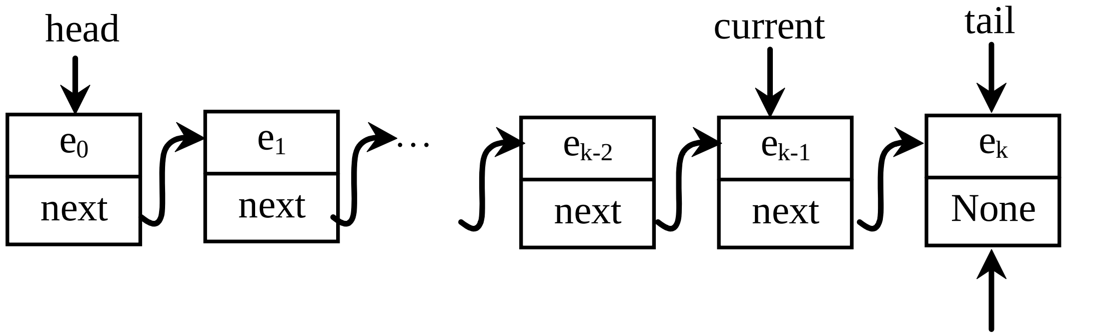
(a) Before the node is deleted.



(b) After the node is deleted.

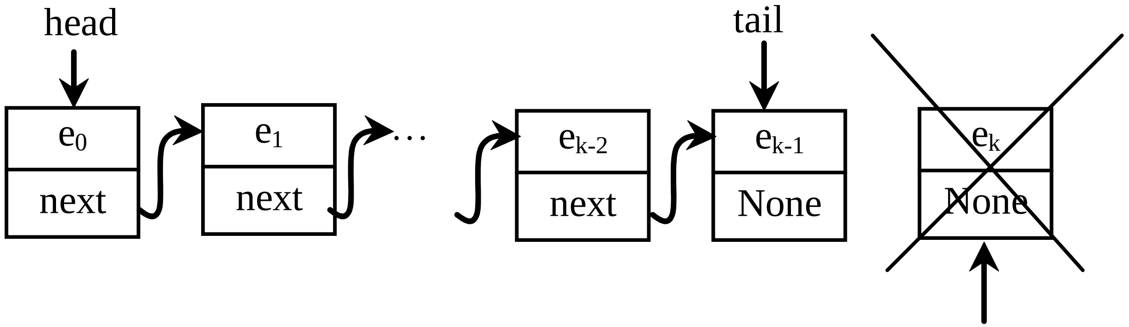
Implementing removeLast()

```
def removeLast(self):
    if self.__size == 0:
        return None
    elif self.__size == 1:
        temp = self.__head
        self.__head = self.__tail = None
        self.__size = 0
        return temp.element
    else:
        current = self.__head

        for i in range(self.__size - 1):
            current = current.next

        temp = self.__tail
        self.__tail = current
        self.__tail.next = None
        self.__size -= 1
        return temp.element
```



Delete this node

(a) Before the node is deleted.

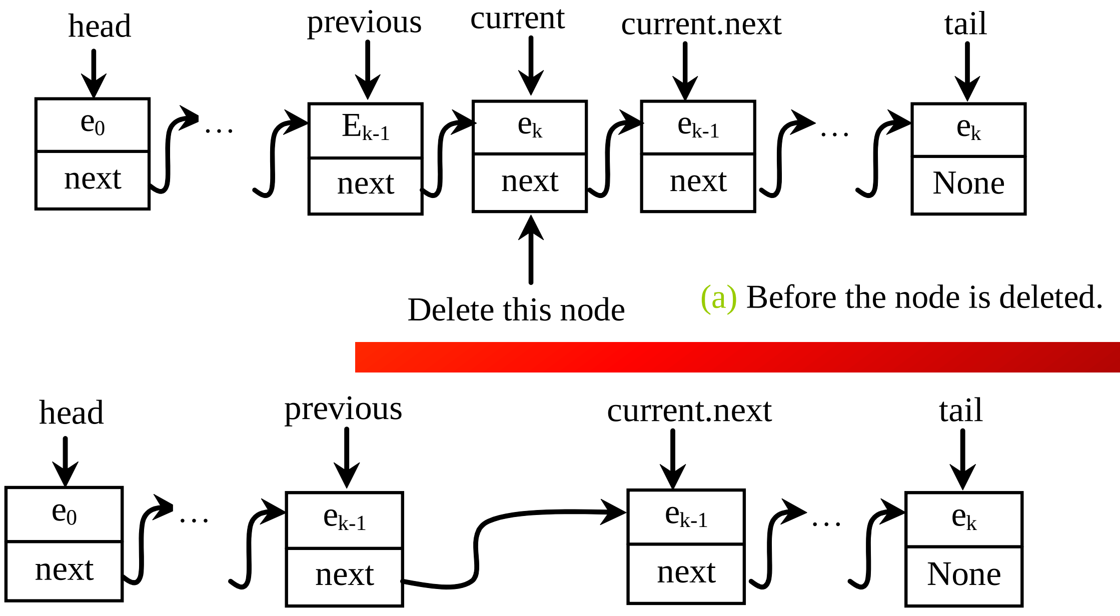


This node is deleted

(b) After the node is deleted.

Implementing removeAt(index)

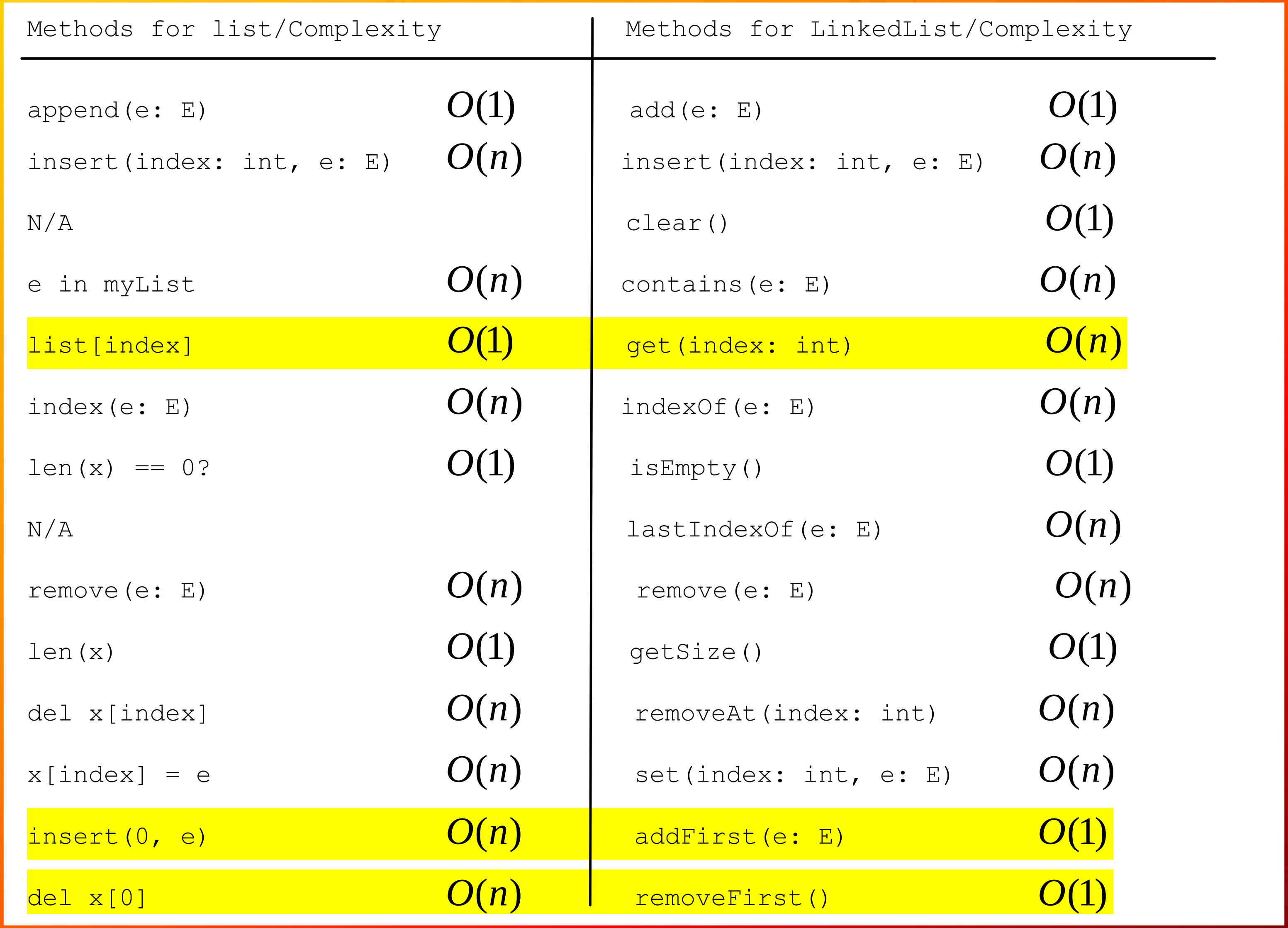
```
def removeAt(self, index):
    if index ==0 | | index>= self.__size:
        return None # Out of range
    elif index == 0:
        return self.removeFirst() # Remove first
    elif index == self.__size - 1:
        return self.removeLast() # Remove last
    else:
        previous = self.__head

        for i in range(index):
            previous = previous.next

        current = previous.next
        previous.next = current.next
        self.__size -= 1
        return current.element
```



(a) Before the node is deleted.

(b) After the node is deleted.

Time Complexity for list and LinkedList



Methods for list/Complexity		Methods for LinkedList/Complexity	
append(e: E)	$O(1)$	add(e: E)	$O(1)$
insert(index: int, e: E)	$O(n)$	insert(index: int, e: E)	$O(n)$
N/A		clear()	$O(1)$
e in myList	$O(n)$	contains(e: E)	$O(n)$
list[index]	$O(1)$	get(index: int)	$O(n)$
index(e: E)	$O(n)$	indexOf(e: E)	$O(n)$
len(x) == 0?	$O(1)$	isEmpty()	$O(1)$
N/A		lastIndexOf(e: E)	$O(n)$
remove(e: E)	$O(n)$	remove(e: E)	$O(n)$
len(x)	$O(1)$	getSize()	$O(1)$
del x[index]	$O(n)$	removeAt(index: int)	$O(n)$
x[index] = e	$O(n)$	set(index: int, e: E)	$O(n)$
insert(0, e)	$O(n)$	addFirst(e: E)	$O(1)$
del x[0]	$O(n)$	removeFirst()	$O(1)$

Ordered Linked List Operations

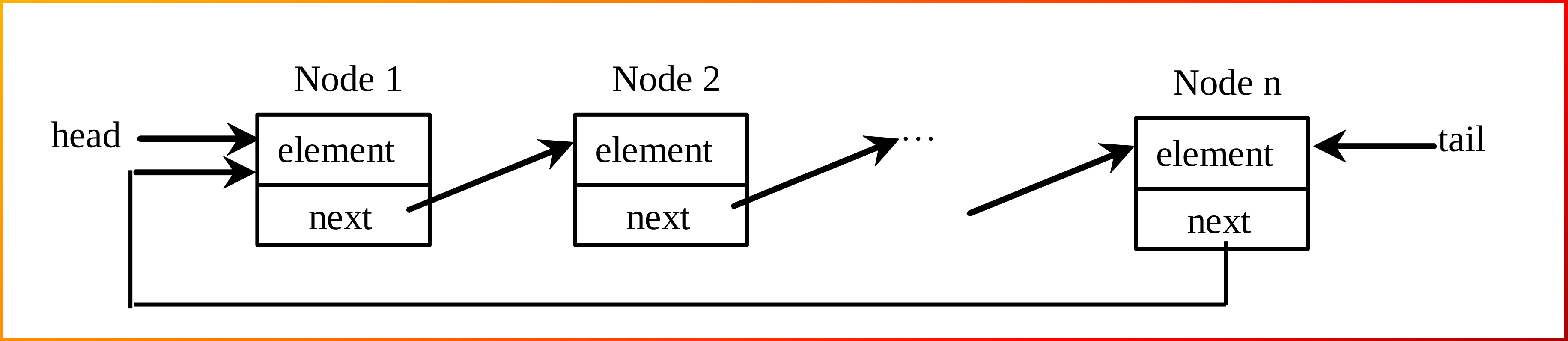
- OrderedList() creates a new ordered list that is empty. It needs no parameters and returns an empty list.
- add(item) adds a new item to the list making sure that the order is preserved. It needs the item and returns nothing. Assume the item is not already in the list.
- remove(item) removes the item from the list. It needs the item and modifies the list. Assume the item is present in the list.
- search(item) searches for the item in the list. It needs the item and returns a boolean value.
- isEmpty() tests to see whether the list is empty. It needs no parameters and returns a Boolean value.
- size() returns the number of items in the list. It needs no parameters and returns an integer.
- index(item) returns the position of item in the list. It needs the item and returns the index. Assume the item is in the list.
- pop() removes and returns the last item in the list. It needs nothing and returns an item. Assume the list has at least one item.
- pop(pos) removes and returns the item at position pos. It needs the position and returns the item. Assume the item is in the list.

Unordered Linked List Operations

- List() creates a new list that is empty. It needs no parameters and returns an empty list.
- add(item) adds a new item to the list. It needs the item and returns nothing. Assume the item is not already in the list.
- remove(item) removes the item from the list. It needs the item and modifies the list. Assume the item is present in the list.
- search(item) searches for the item in the list. It needs the item and returns a Boolean value.
- isEmpty() tests to see whether the list is empty. It needs no parameters and returns a Boolean value.
- size() returns the number of items in the list. It needs no parameters and returns an integer.
- append(item) adds a new item to the end of the list making it the last item in the collection. It needs the item and returns nothing. Assume the item is not already in the list.
- index(item) returns the position of item in the list. It needs the item and returns the index. Assume the item is in the list.
- insert(pos,item) adds a new item to the list at position pos. It needs the item and returns nothing. Assume the item is not already in the list and there are enough existing items to have position pos.
- pop() removes and returns the last item in the list. It needs nothing and returns an item. Assume the list has at least one item.
- pop(pos) removes and returns the item at position pos. It needs the position and returns the item. Assume the item is in the list.

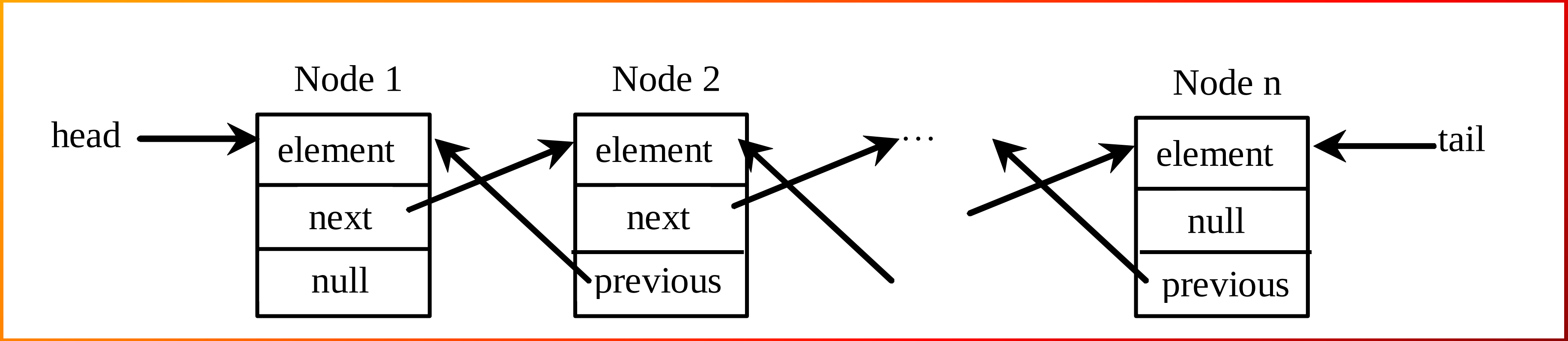
Circular Linked Lists

- A *circular, singly linked list* is like a singly linked list, except that the pointer of the last node points back to the first node.



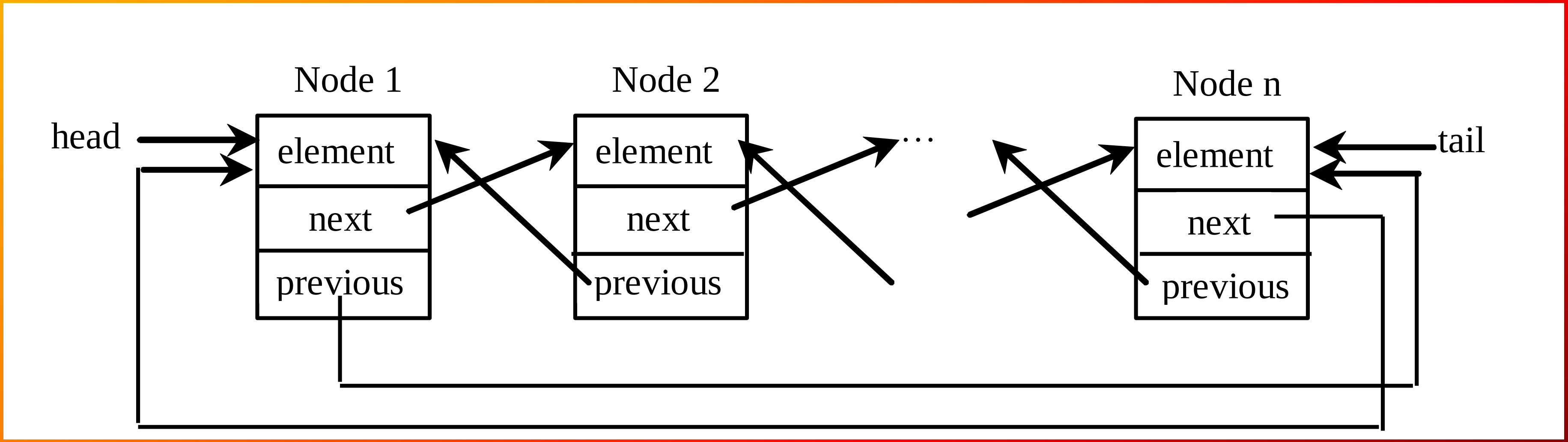
Doubly Linked Lists

- A *doubly linked list* contains the nodes with two pointers. One points to the next node and the other points to the previous node. These two pointers are conveniently called *a forward pointer* and *a backward pointer*. So, a doubly linked list can be traversed forward and backward.



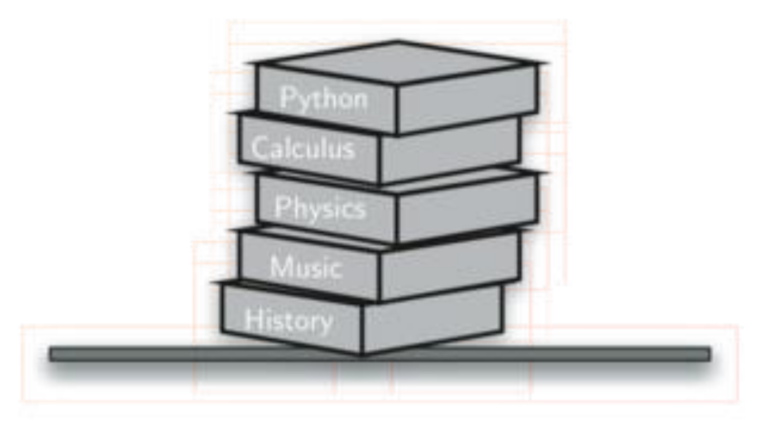
Circular Doubly Linked Lists

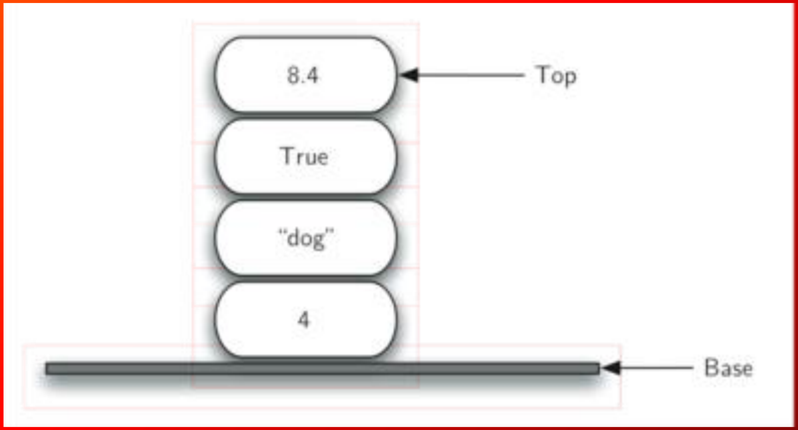
- A *circular, doubly linked list* is doubly linked list, except that the forward pointer of the last node points to the first node and the backward pointer of the first pointer points to the last node.



Stack

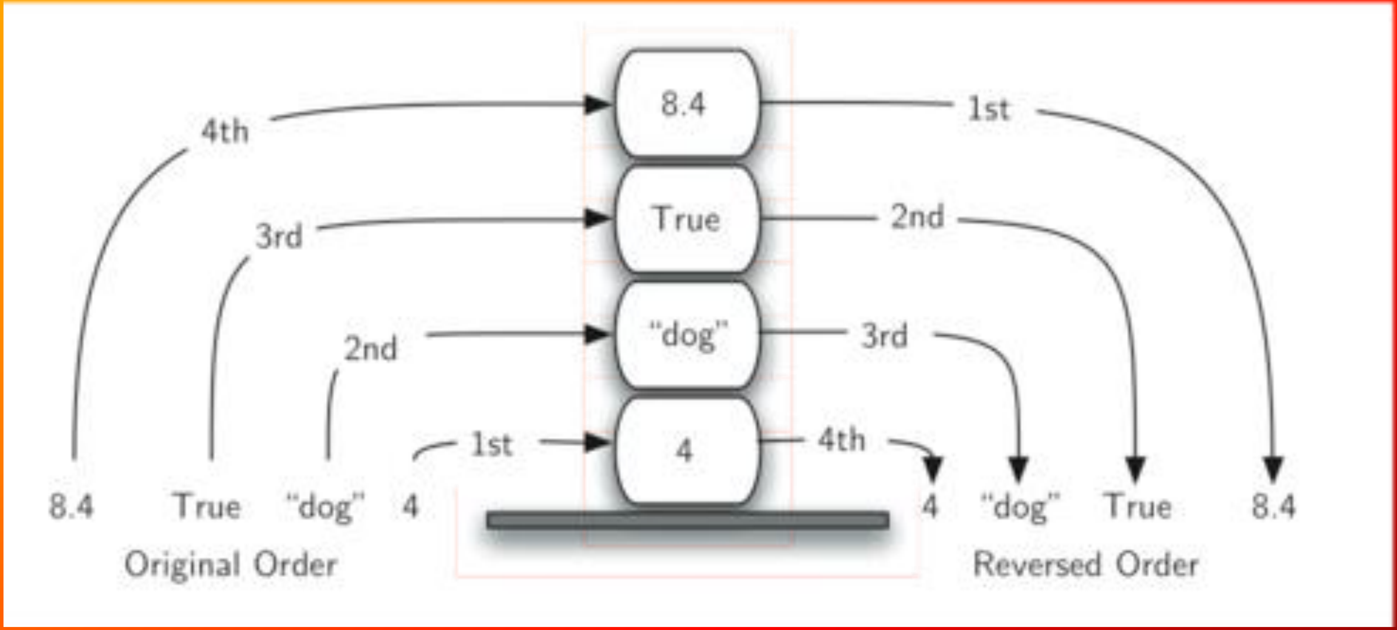
- is an ordered collection of items where the addition of new items and the removal of existing items always takes place at the same end. This end is commonly referred to as the "top." The end opposite the top is known as the "base."





Order in the Stack

- The most recently added item is the one that is in position to be removed first. This ordering principle is sometimes called **LIFO**, **last-in first-out**.



Stack Operations

- Stack() creates a new stack that is empty. It needs no parameters and returns an empty stack.

- push(item) adds a new item to the top of the stack. It needs the item and returns nothing.

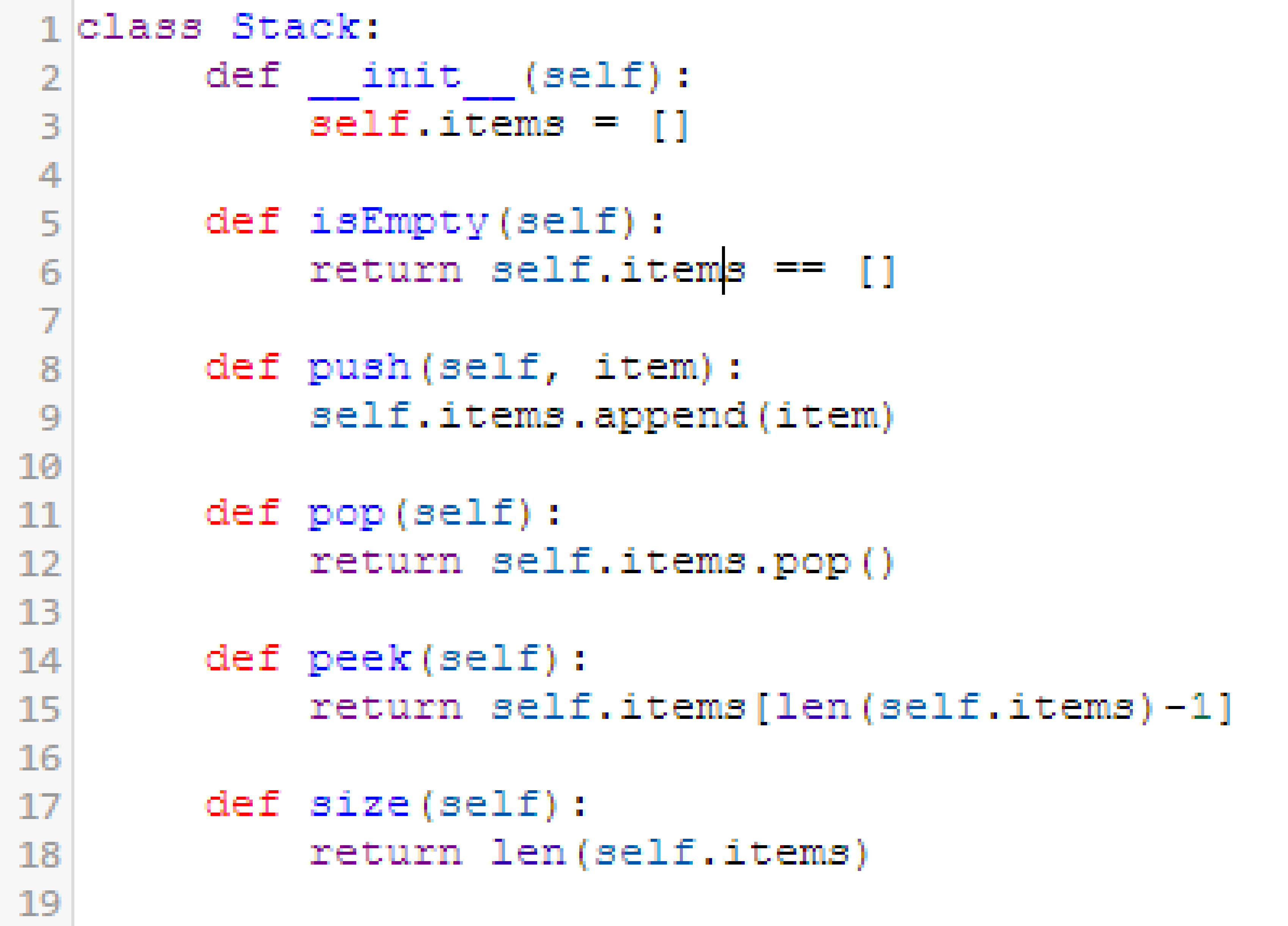
- pop() removes the top item from the stack. It needs no parameters and returns the item. The stack is modified.

- peek() returns the top item from the stack but does not remove it. It needs no parameters. The stack is not modified.

- isEmpty() tests to see whether the stack is empty. It needs no parameters and returns a Boolean value.

- size() returns the number of items on the stack. It needs no parameters and returns an integer

Stack Implementation



```
class Stack:
     def __init__(self):
         self.items = []

     def isEmpty(self):
         return self.items == []

     def push(self, item):
         self.items.append(item)

     def pop(self):
         return self.items.pop()

     def peek(self):
         return self.items[len(self.items)-1]

     def size(self):
         return len(self.items)
```

Stack Example

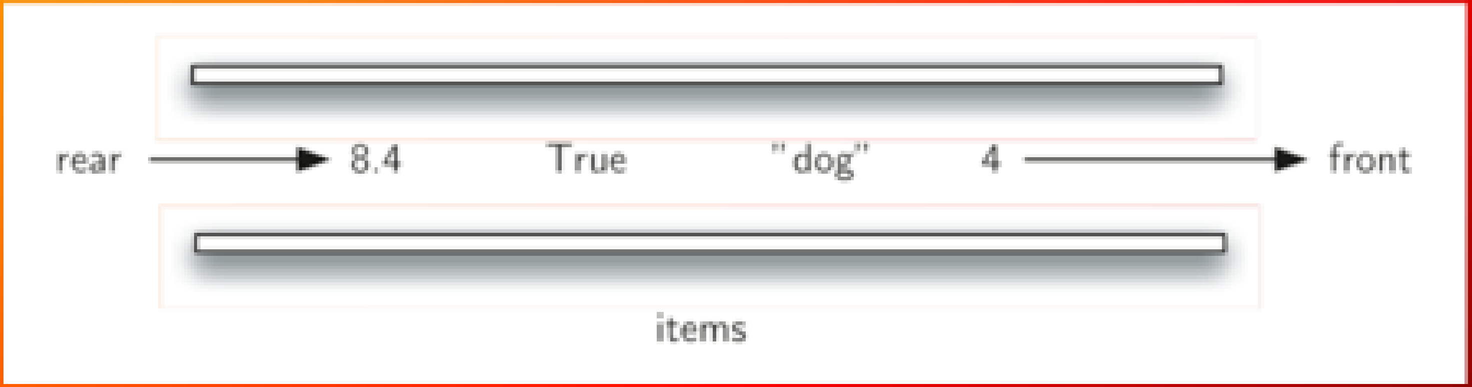
Stack Operation	Stack Contents	Return Value
s.isEmpty()	[]	True
s.push(4)	[4]	
s.push('dog')	[4,'dog']	
s.peek()	[4,'dog']	'dog'
s.push(True)	[4,'dog',True]	
s.size()	[4,'dog',True]	3
s.isEmpty()	[4,'dog',True]	False
s.push(8.4)	[4,'dog',True,8.4]	
s.pop()	[4,'dog',True]	8.4
s.pop()	[4,'dog']	True
s.size()	[4,'dog']	2

Queue

- A queue is an ordered collection of items where the addition of new items happens at one end, called the “rear,” and the removal of existing items occurs at the other end, commonly called the “front.”
- As an element enters the queue it starts at the rear and makes its way toward the front, waiting until that time when it is the next element to be removed.

Order in Queue

- The most recently added item in the queue must wait at the end of the collection. The item that has been in the collection the longest is at the front. This ordering principle is sometimes called **FIFO**, **first-in first-out**. It is also known as "first-come first-served."



Queue Operations

- Queue() creates a new queue that is empty. It needs no parameters and returns an empty queue.

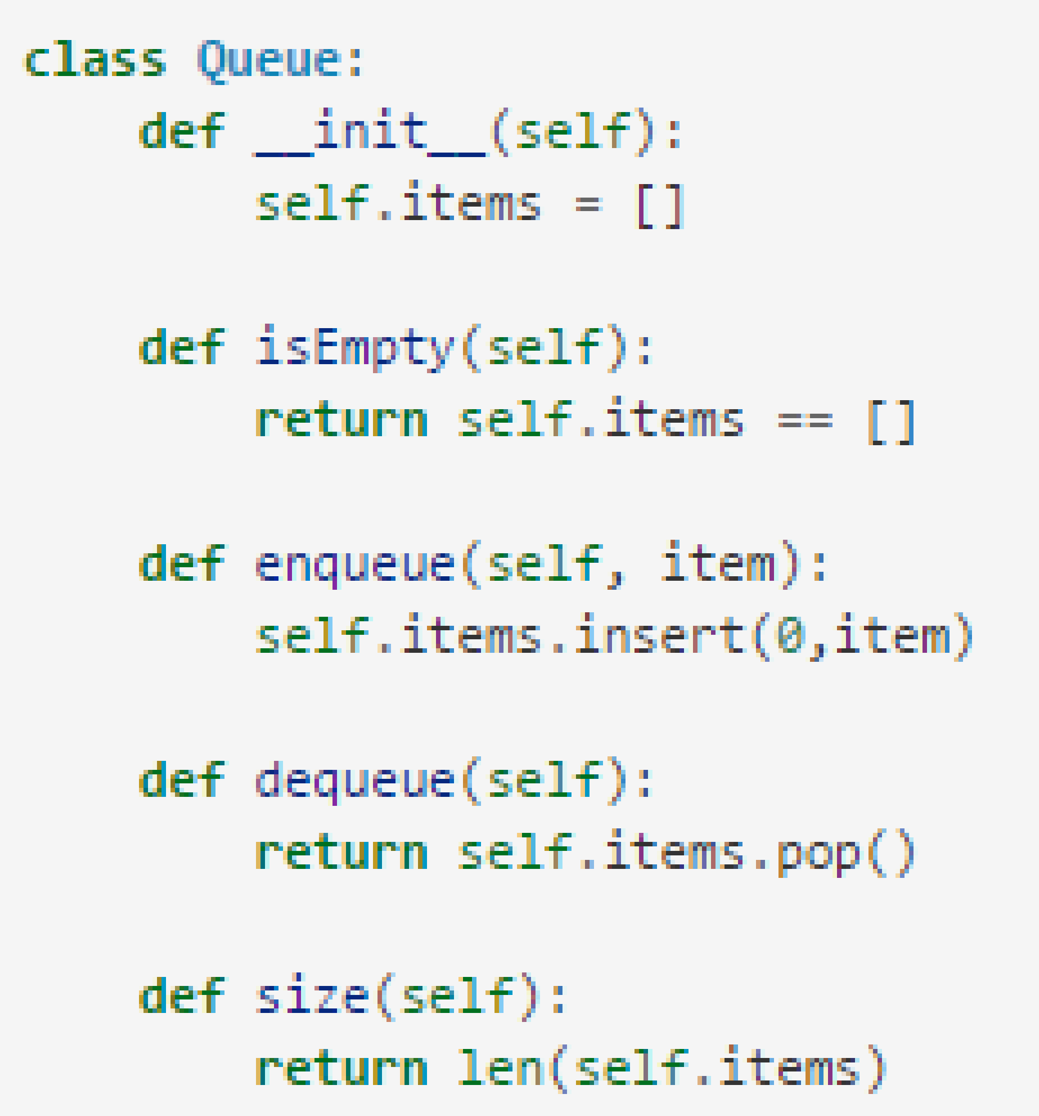
- enqueue(item) adds a new item to the rear of the queue. It needs the item and returns nothing.

- dequeue() removes the front item from the queue. It needs no parameters and returns the item. The queue is modified.

- isEmpty() tests to see whether the queue is empty. It needs no parameters and returns a Boolean value.

- size() returns the number of items in the queue. It needs no parameters and returns an integer.

Queue Implementation



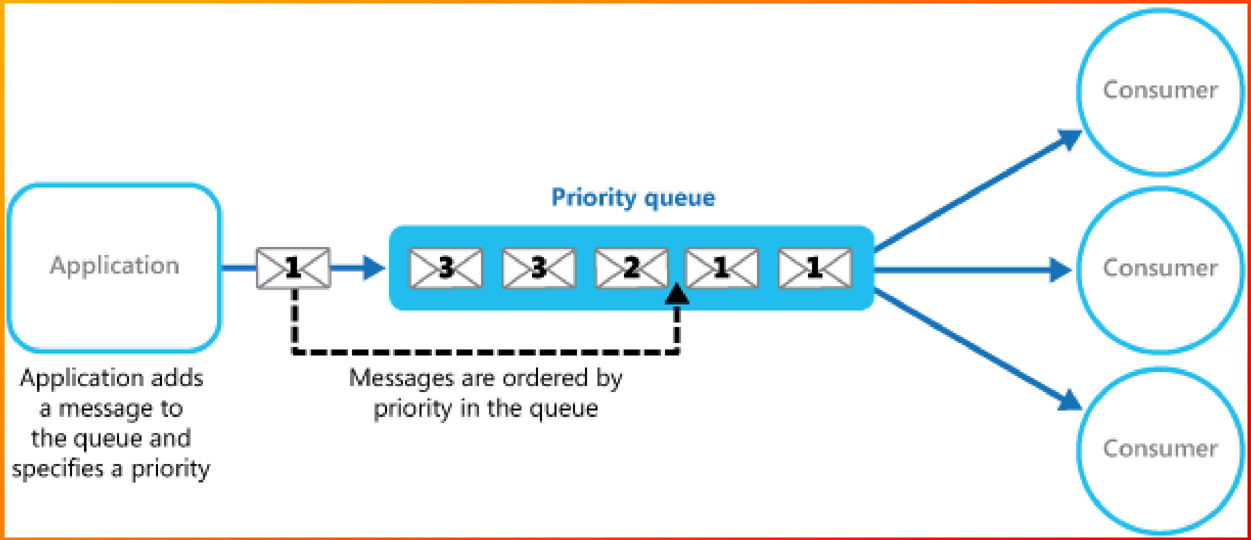
```
class Queue:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def enqueue(self, item):
        self.items.insert(0,item)

    def dequeue(self):
        return self.items.pop()

    def size(self):
        return len(self.items)
```

Queue Example

Queue Operation	Queue Contents	Return Value
q.isEmpty()	[]	True
q.enqueue(4)	[4]	
q.enqueue('dog')	['dog',4]	
q.enqueue(True)	[True,'dog',4]	
q.size()	[True,'dog',4]	3
q.isEmpty()	[True,'dog',4]	False
q.enqueue(8.4)	[8.4,True,'dog',4]	
q.dequeue()	[8.4,True,'dog']	4
q.dequeue()	[8.4,True]	'dog'
q.size()	[8.4,True]	2

Priority Queue

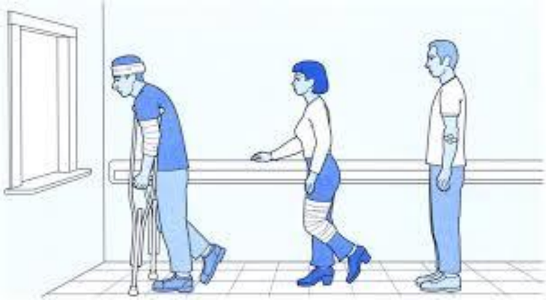
- In a priority queue, elements are assigned with priorities. When accessing elements, the element with the highest priority is removed first. A priority queue has a largest-in, first-out behavior.



Application adds a message to the queue and specifies a priority

Messages are ordered by priority in the queue

- For example, the emergency room in a hospital assigns patients with priority numbers; the patient with the highest priority is treated first.



Any Questions?