INDUSTRIAL UNIVERSITY OF HOCHIMINH CITY

Linked Lists

Data Structures and Algorithms with Python

Lecture 7

Overview

- Introduction
- Various types of Linked-Lists
  - Singly Linked Lists
  - Circularly Linked Lists
  - Doubly Linked Lists
- The positional list ADT
- Sorting a positional list
- Case study: Maintaining Access Frequencies
- Link-Based Vs Array-based Sequences
- Summary

Introduction: Arrays Vs LinkedLists

Array-based Sequences

- An ***array*** provides more ***centralized representation***, with one large chunk of memory capable of accommodating references to many elements.

Limitations of Array-based sequences:

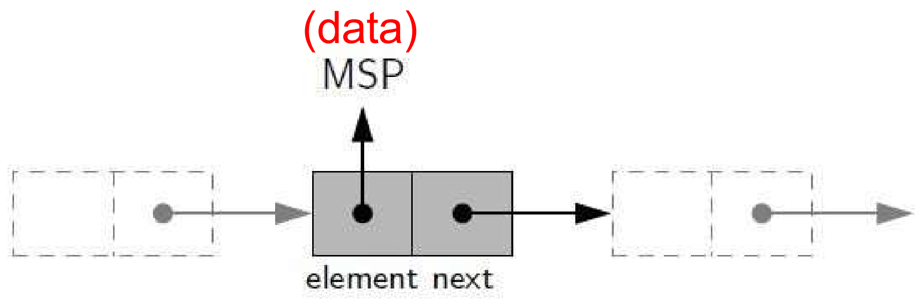
- The length of a dynamic array might be longer than the actual number of elements that it stores.
- Amortized bounds for operations may be unacceptable in real-time systems.
- Insertions and deletions at interior positions of an array are expensive.

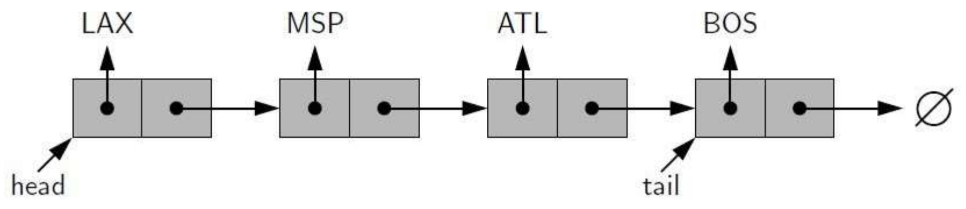
- ***Linked-lists*** is an alternative to an array-based sequence which overcomes the these limitations.

- It relies on a more ***distributed representation*** in which a lightweight object, known as a ***node***, is allocated for each element.
  - Each node maintains a reference to its element and one or more references to neighboring nodes in order to collectively represent the linear order of the sequence.
- Linked-lists has the following **limitations**:
  - Elements of a linked list cannot be efficiently accessed by a numeric index k.
  - We cannot tell just by examining a node if it is the second, fifth, or twentieth node in the list.

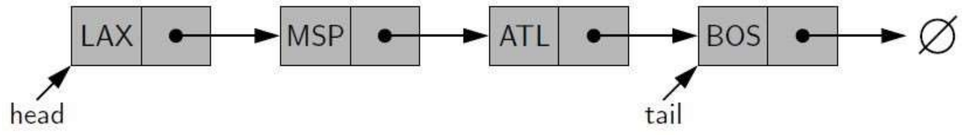
Singly Linked Lists

- A singly linked list, in its simplest form, is a collection of ***nodes*** that collectively form a linear sequence.
- Each ***node*** stores a reference to an object that is an element of the sequence, as well as a reference to the next node of the list.
- The first and last node of a linked list are known as the ***head*** and ***tail*** of the list, respectively.
- By starting at the head, and moving from one node to another by following each node's next reference, we can reach the tail of the list - ***traversing*** the linked list.
- We can identify the ***tail*** as the node having None as its next reference.
- Because the next reference of a node can be viewed as a link or pointer to another node, the process of traversing a list is also known as ***link hopping*** or ***pointer hopping***.





Singly linked list for storing airport codes



Simplified view

Minimally, the linked list instance should keep:

- *A reference to the head of the list*. Without an explicit reference to the head, there would be no way to locate that node (or indirectly, any others).

- *An explicit reference to the tail node* (Not required, but preferred to prevent traversing the rest of the list).

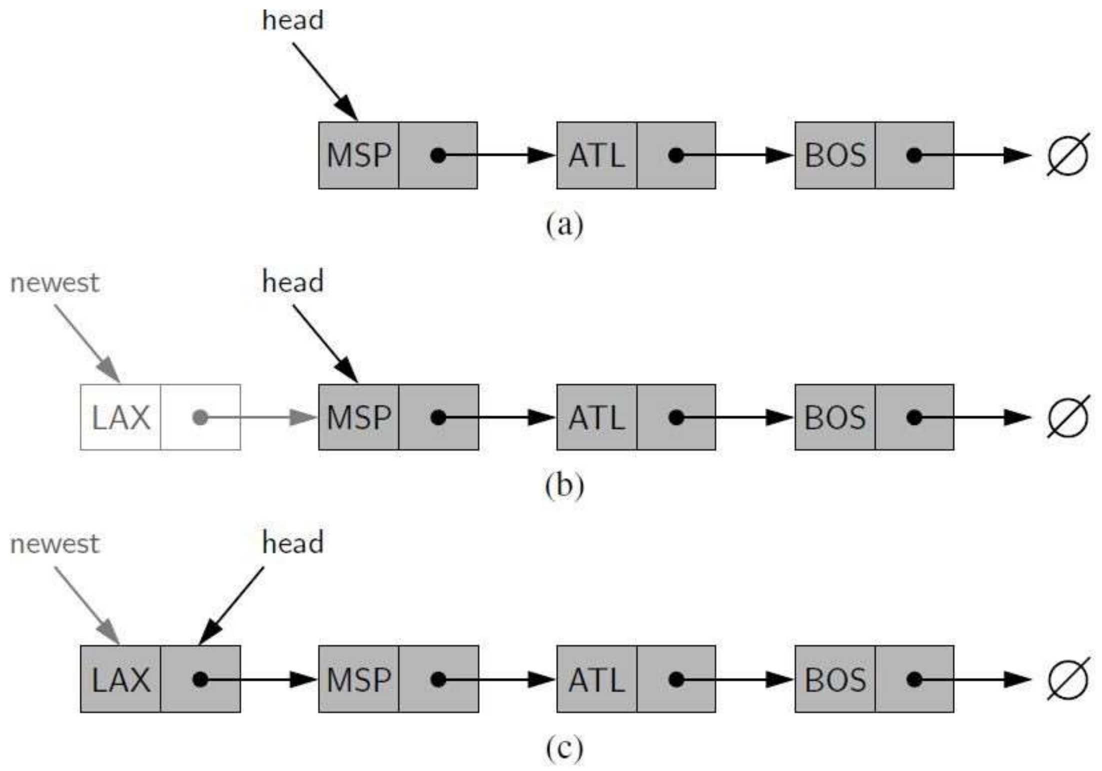
- *Count of the total number of nodes* that comprise the list - size (Not required, preferred to avoid traversing the list to get the count).

Inserting an Element at the Head of a Singly Linked List

- An important property of a linked list is that it does not have a predetermined fixed size; it uses space proportionally to its current number of elements.

- When using a singly linked list, we can easily insert an element at the head of the list.

- Steps for inserting an element:
    - Create a new node.
    - Set its element to the new element
    - Set its next link to refer to the current head
    - Set the list's head to point to the new node.

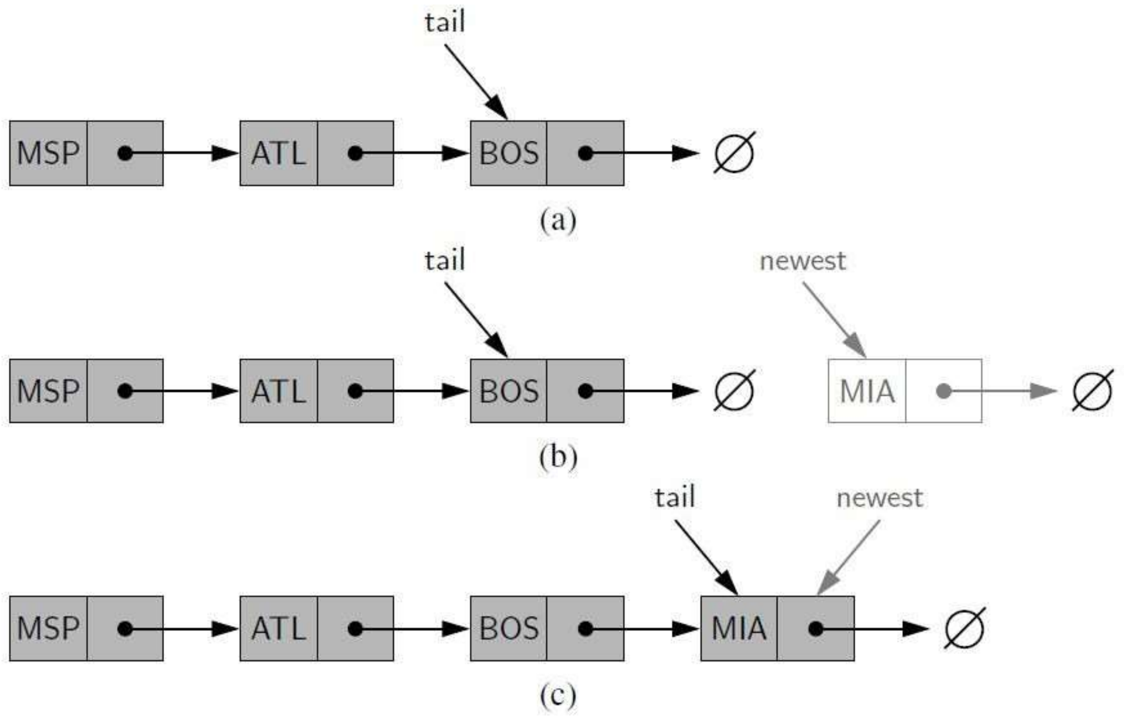


```
Algorithm add_first(L,e):
    newest = Node(e)  {create new node instance storing reference to element e}
    newest.next = L.head    {set new node's next to reference the old head node}
    L.head = newest              {set variable head to reference the new node}
    L.size = L.size + 1                              {increment the node count}
```

Inserting an Element at the Tail of a Singly Linked List

Steps:

- Create a new node
- Assign its next reference to None.
- Set the next reference of the tail to point to this new node
- Update the tail reference itself to this new node.



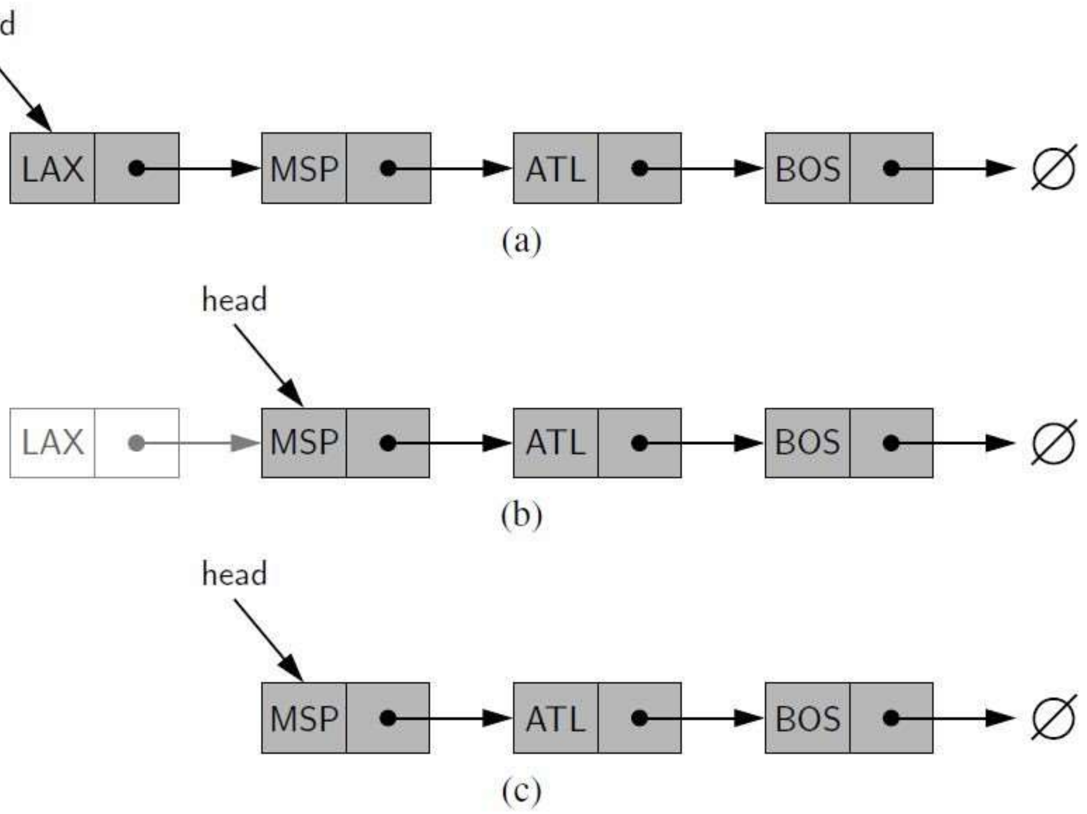
```
Algorithm add_last(L, e):
    newest = Node(e)  {create new node instance storing reference to element e}
    newest.next = None        {set new node's next to reference the None object}
    L.tail.next = newest                {make old tail node point to new node}
    L.tail = newest                 {set variable tail to reference the new node}
    L.size = L.size + 1                               {increment the node count}
```

Removing an Element from a Singly Linked List

- ***Removing an element from the head*** of a singly linked list is essentially the reverse operation of inserting a new element at the head.

- Unfortunately it is ***not easy to delete the last node*** of a singly linked list.
  - Even if we maintain a tail reference directly to the last node of the list, we must be able to access the ***node before the last node*** in order to remove the last node.
  - We cannot reach the node before the tail by following next links from the tail.
  - The only way to access this node is to start from the head of the list and search all the way through the list.
  - Such a sequence of link-hopping operations could take a long time.

- This operation is carried more efficiently in ***doubly linked list.***



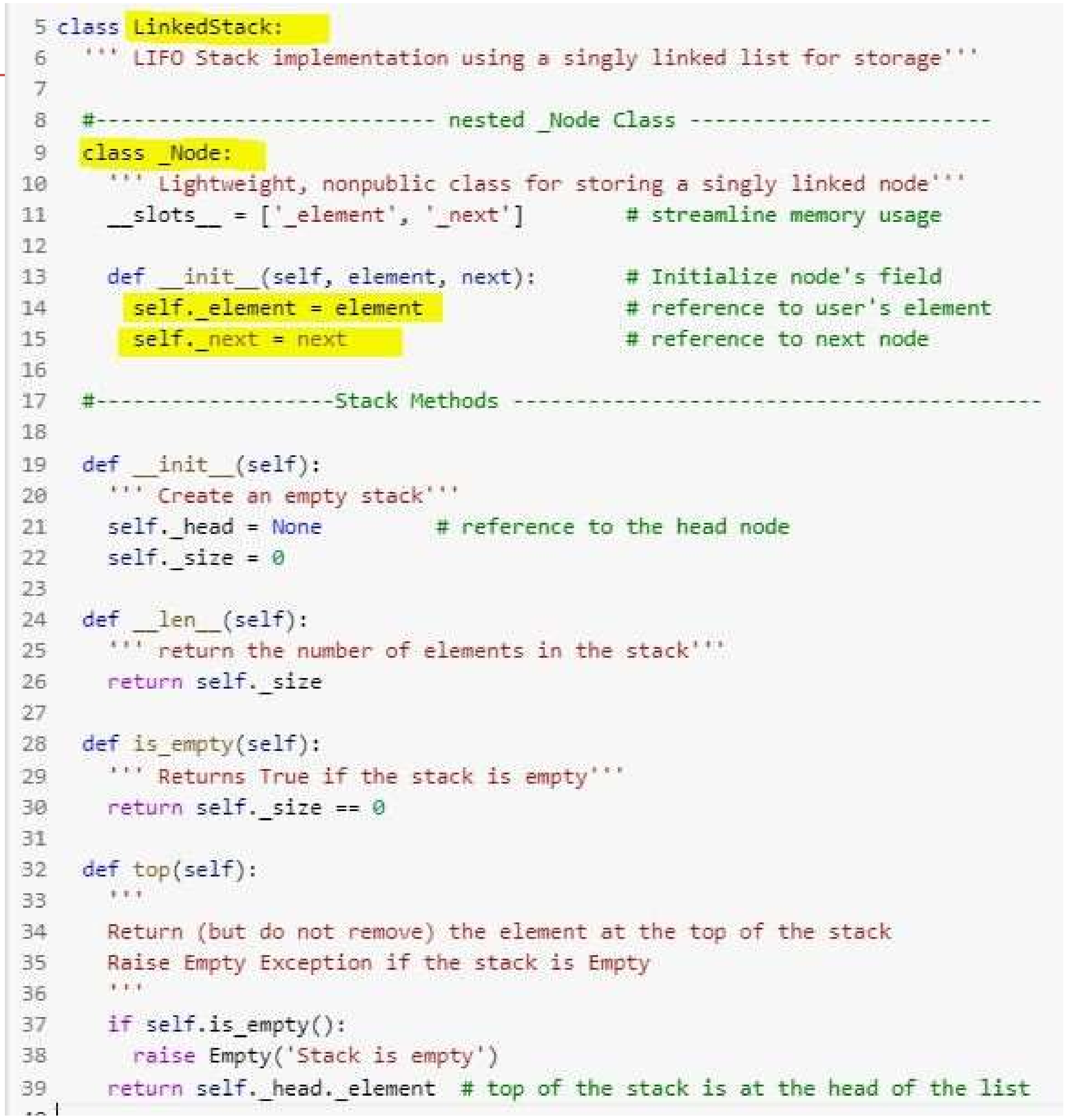
```
Algorithm remove_first(L):
    if L.head is None then
        Indicate an error: the list is empty.
    L.head = L.head.next          {make head point to next node (or None)}
    L.size = L.size − 1                      {decrement the node count}
```

Implementing a Stack with a Singly Linked List

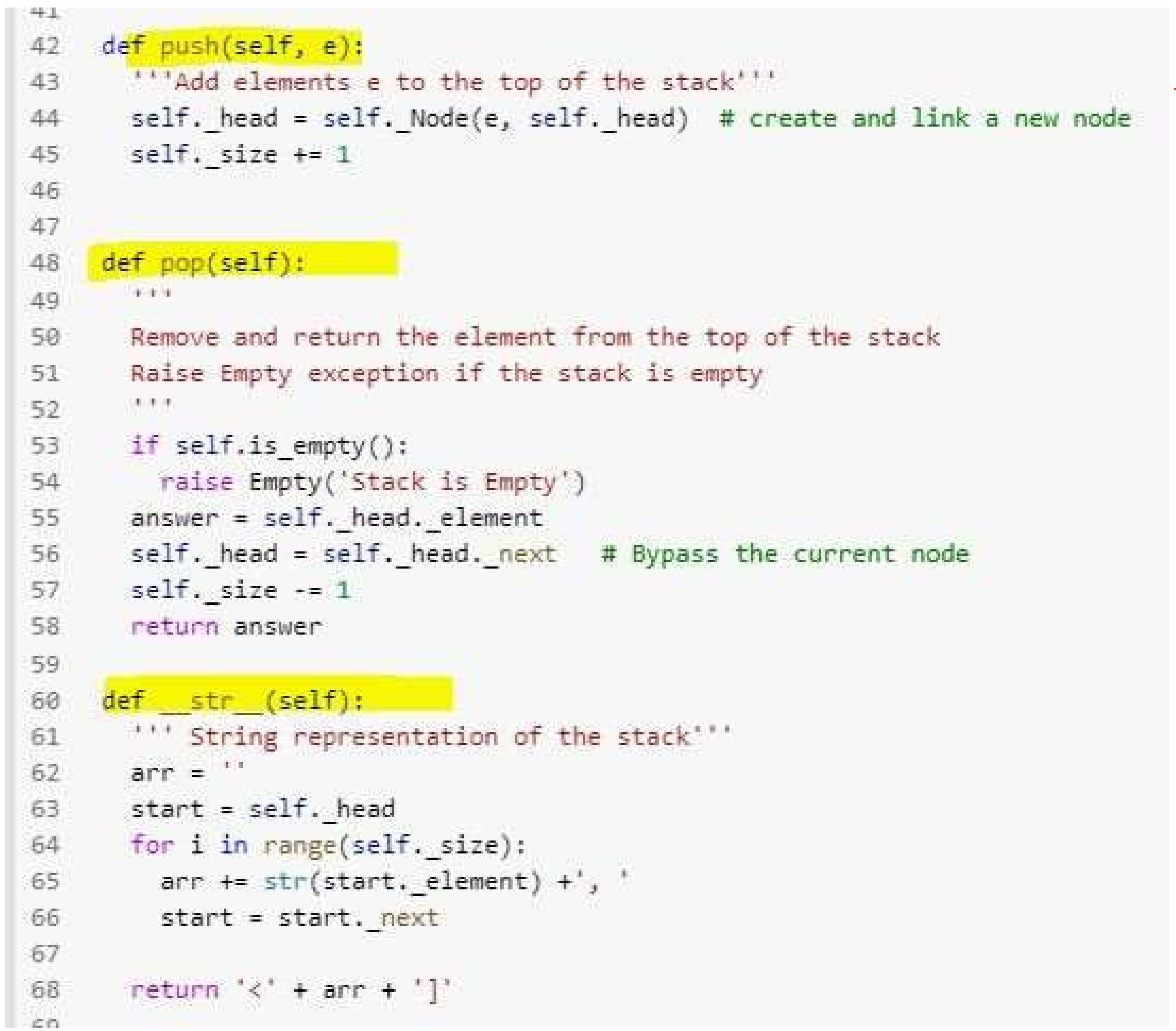
- This implementation of Stack maintains two variables:
  - **`_head`**: reference to the node at the head of the list
  - **`_size`**: to keep track of the current number of elements in the stack.

- The stack uses a nonpublic class **`_Node`** to represent each node which are pushed or popped into or out of the stack.

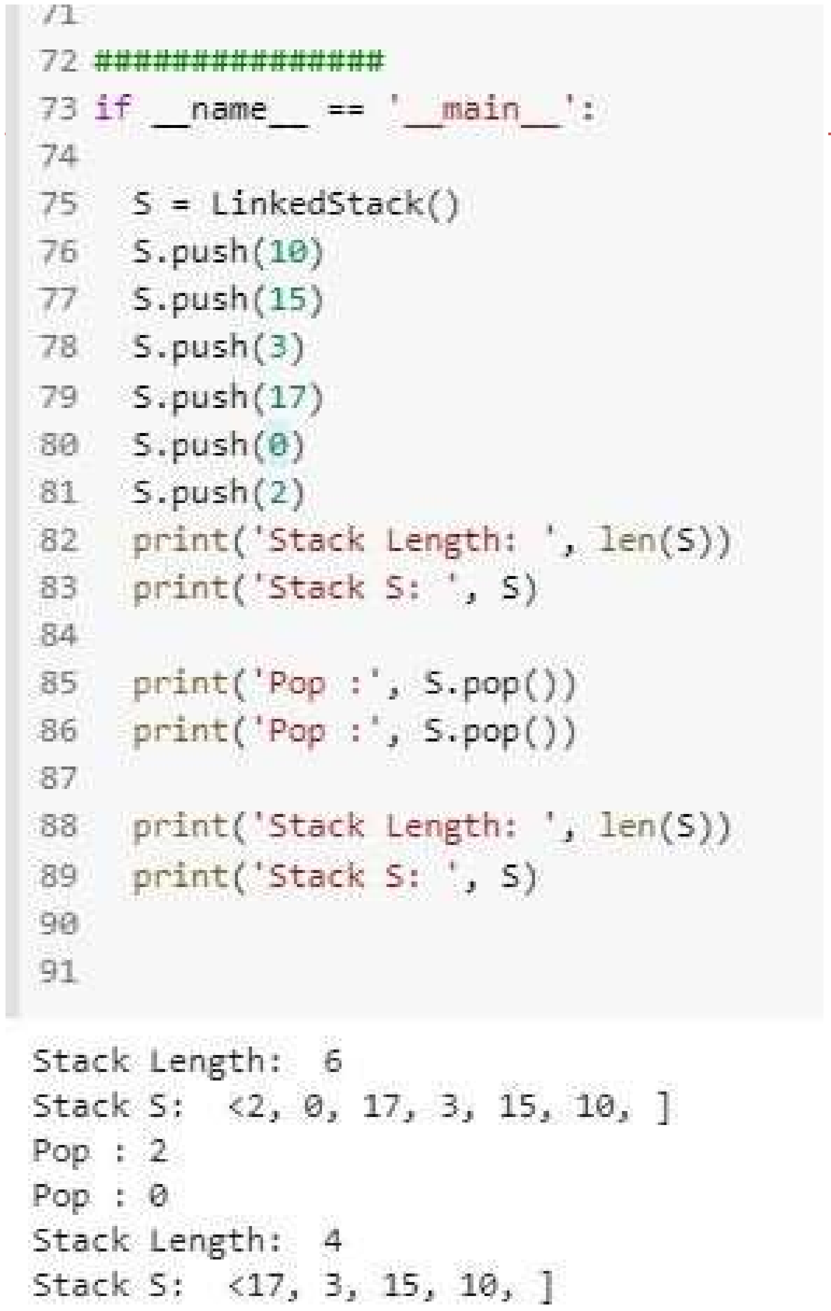
- The **`_Node`** object maintains two variables:
  - **`_element`**: to store data
  - **`_next`**: reference to next node



```
class LinkedStack:
  ''' LIFO Stack implementation using a singly linked list for storage'''

  #------------------------- nested _Node Class ----------------------
  class _Node:
    ''' Lightweight, nonpublic class for storing a singly linked node'''
    __slots__ = ['_element', '_next']         # streamline memory usage

    def __init__(self, element, next):        # Initialize node's field
      self._element = element                 # reference to user's element
      self._next = next                       # reference to next node

  #-----------------Stack Methods ------------------------------------

  def __init__(self):
    ''' Create an empty stack'''
    self._head = None           # reference to the head node
    self._size = 0

  def __len__(self):
    ''' return the number of elements in the stack'''
    return self._size

  def is_empty(self):
    ''' Returns True if the stack is empty'''
    return self._size == 0

  def top(self):
    '''
    Return (but do not remove) the element at the top of the stack
    Raise Empty Exception if the stack is Empty
    '''
    if self.is_empty():
      raise Empty('Stack is empty')
    return self._head._element  # top of the stack is at the head of the list
```



```
  def push(self, e):
    '''Add elements e to the top of the stack'''
    self._head = self._Node(e, self._head)  # create and link a new node
    self._size += 1


  def pop(self):
    '''
    Remove and return the element from the top of the stack
    Raise Empty exception if the stack is empty
    '''
    if self.is_empty():
      raise Empty('Stack is Empty')
    answer = self._head._element
    self._head = self._head._next   # Bypass the current node
    self._size -= 1
    return answer

  def __str__(self):
    ''' String representation of the stack'''
    arr = ''
    start = self._head
    for i in range(self._size):
      arr += str(start._element) +', '
      start = start._next

    return '<' + arr + ']'
```



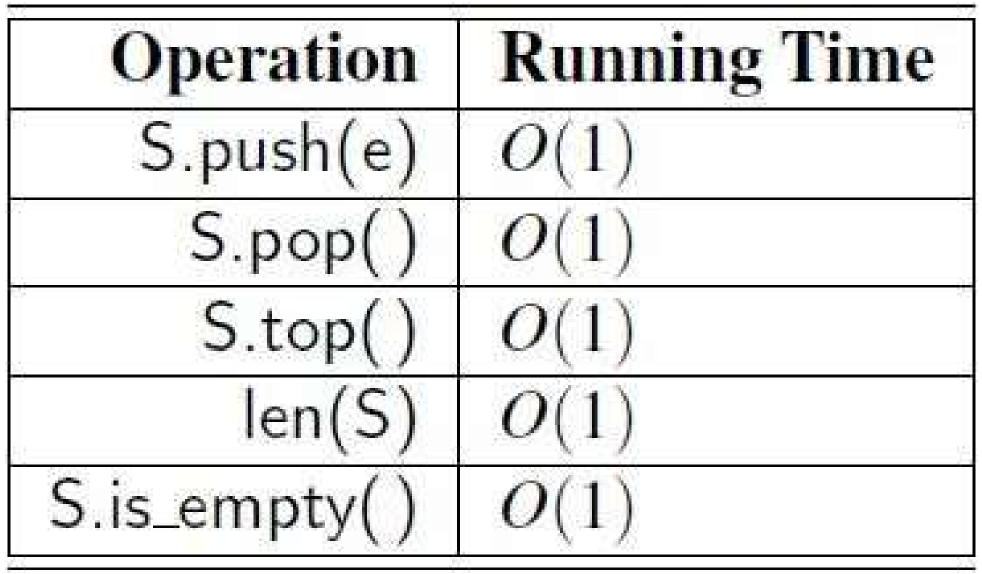
```
################
if __name__ == '__main__':

  S = LinkedStack()
  S.push(10)
  S.push(15)
  S.push(3)
  S.push(17)
  S.push(0)
  S.push(2)
  print('Stack Length: ', len(S))
  print('Stack S: ', S)

  print('Pop :', S.pop())
  print('Pop :', S.pop())

  print('Stack Length: ', len(S))
  print('Stack S: ', S)


```

```
Stack Length:  6
Stack S:  <2, 0, 17, 3, 15, 10, ]
Pop : 2
Pop : 0
Stack Length:  4
Stack S:  <17, 3, 15, 10, ]
```

Operations

Analysis of `LinkedStack`

- All of the methods complete in worst-case constant time.
- No amortized analysis is required in this case.
- Space usage is O(n) where n is the current number of elements in the stack.



Operation	Running Time
S.push(e)	$O(1)$
S.pop()	$O(1)$
S.top()	$O(1)$
len(S)	$O(1)$
S.is_empty()	$O(1)$

Implementing a Queue with a Singly Linked List

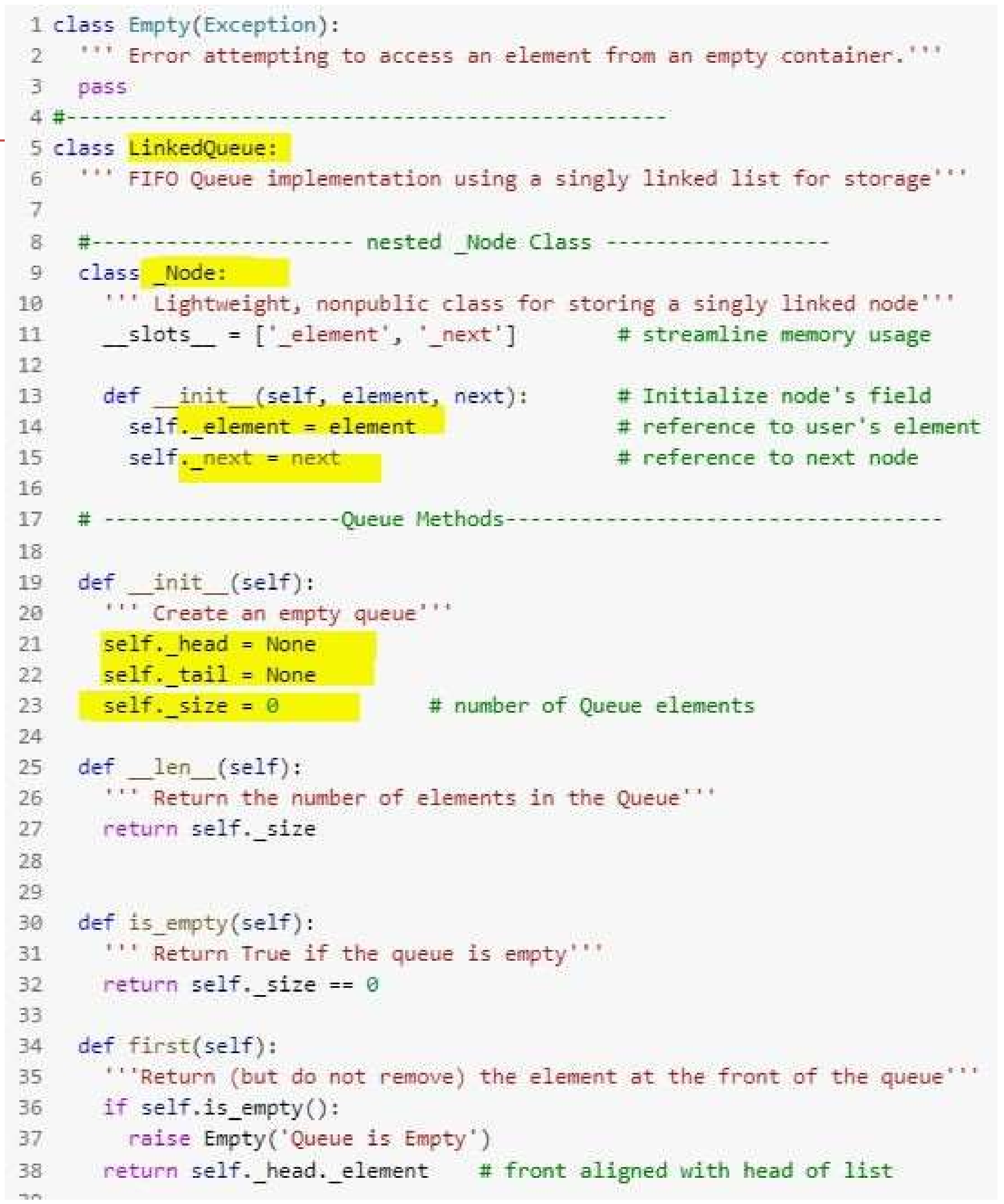
- This ensures worst-case O(1) time for all operations.

- This implementation maintains three instance variables:
  - **`_head`**, **`_tail`** and **`_size`** to avoid traversing the list.

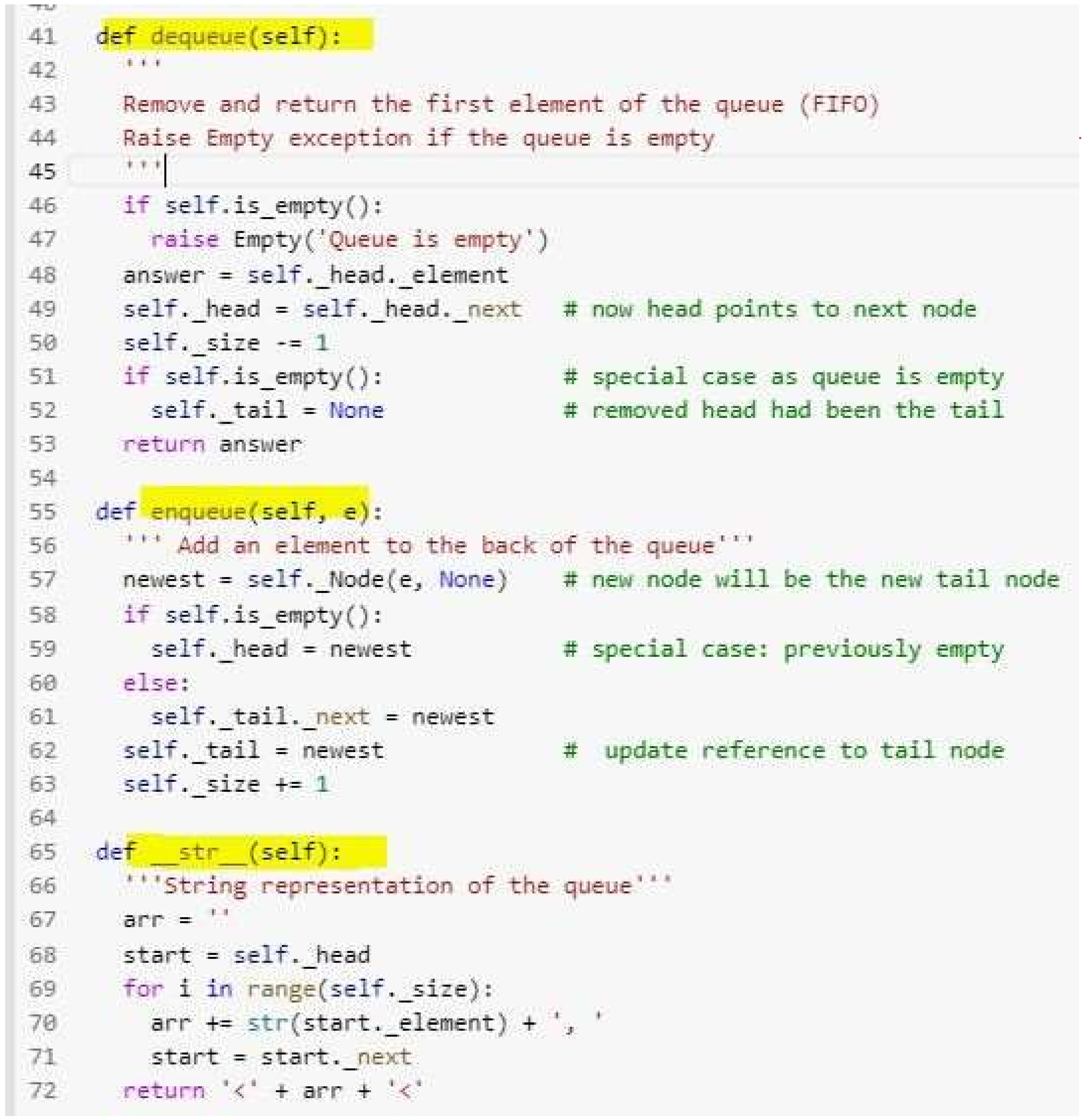
- Elements are added (enqueued) to the end of the list and removed (dequeued) from the front of the list.

- Additional care is to be taken to maintain accurate **`_tail`** reference.
  - During enqueue, newest node always becomes the new tail.
  - If there is only one element in the list, it also becomes the new head.

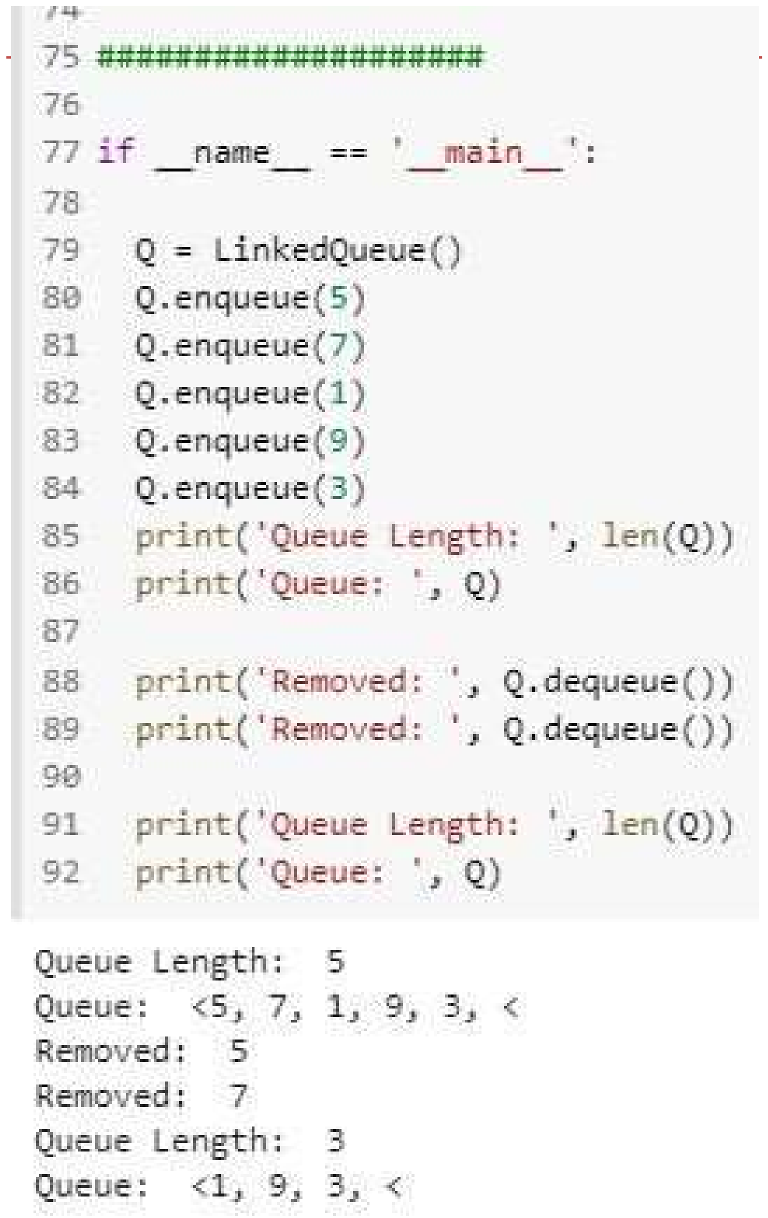
- Space usage is linear in the current number of elements ~ O(n).



```
class Empty(Exception):
  ''' Error attempting to access an element from an empty container.'''
  pass
#---------------------------------------------------------
class LinkedQueue:
  ''' FIFO Queue implementation using a singly linked list for storage'''

  #-------------------- nested _Node Class -----------------
  class _Node:
    ''' Lightweight, nonpublic class for storing a singly linked node'''
    __slots__ = ['_element', '_next']          # streamline memory usage

    def __init__(self, element, next):         # Initialize node's field
      self._element = element                  # reference to user's element
      self._next = next                        # reference to next node

  # -----------------Queue Methods---------------------------------

  def __init__(self):
    ''' Create an empty queue'''
    self._head = None
    self._tail = None
    self._size = 0                # number of Queue elements

  def __len__(self):
    ''' Return the number of elements in the Queue'''
    return self._size


  def is_empty(self):
    ''' Return True if the queue is empty'''
    return self._size == 0

  def first(self):
    '''Return (but do not remove) the element at the front of the queue'''
    if self.is_empty():
      raise Empty('Queue is Empty')
    return self._head._element    # front aligned with head of list
```



```python
  def dequeue(self):
    '''
    Remove and return the first element of the queue (FIFO)
    Raise Empty exception if the queue is empty
    '''
    if self.is_empty():
      raise Empty('Queue is empty')
    answer = self._head._element
    self._head = self._head._next   # now head points to next node
    self._size -= 1
    if self.is_empty():             # special case as queue is empty
      self._tail = None             # removed head had been the tail
    return answer

  def enqueue(self, e):
    ''' Add an element to the back of the queue'''
    newest = self._Node(e, None)    # new node will be the new tail node
    if self.is_empty():
      self._head = newest           # special case: previously empty
    else:
      self._tail._next = newest
    self._tail = newest             #  update reference to tail node
    self._size += 1

  def __str__(self):
    '''String representation of the queue'''
    arr = ''
    start = self._head
    for i in range(self._size):
      arr += str(start._element) + ', '
      start = start._next
    return '<' + arr + '<'
```



```python
#####################

if __name__ == '__main__':

  Q = LinkedQueue()
  Q.enqueue(5)
  Q.enqueue(7)
  Q.enqueue(1)
  Q.enqueue(9)
  Q.enqueue(3)
  print('Queue Length: ', len(Q))
  print('Queue: ', Q)

  print('Removed: ', Q.dequeue())
  print('Removed: ', Q.dequeue())

  print('Queue Length: ', len(Q))
  print('Queue: ', Q)
```

```
Queue Length:  5
Queue:  <5, 7, 1, 9, 3, <
Removed:  5
Removed:  7
Queue Length:  3
Queue:  <1, 9, 3, <
```

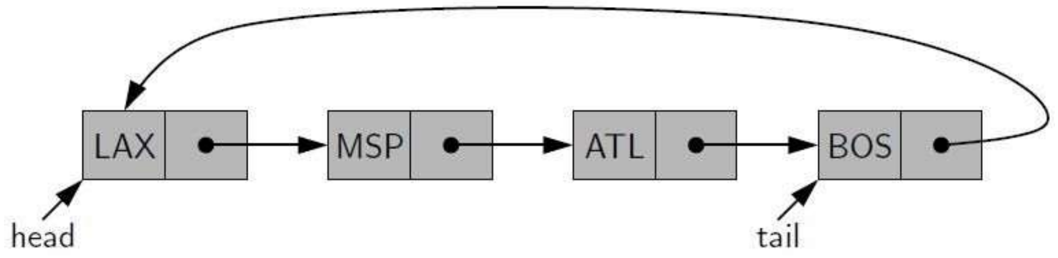
Circularly Linked List

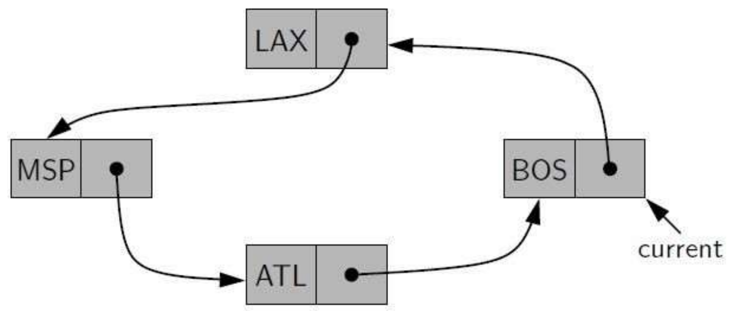
- In the previous chapter, the concept of circular array was implemented using modular arithmetic for advancing an index from the last slot to the first slot of the array.

- In circularly linked list, the tail of the list uses its next reference to point back to the head of the list.

- A circularly linked list provides a more general model than a standard linked list for data sets that are cyclic, that is, which do not have any particular notion of a beginning and end.

- The circularly linked must maintain a reference to particular node to make use of the list - ***current***





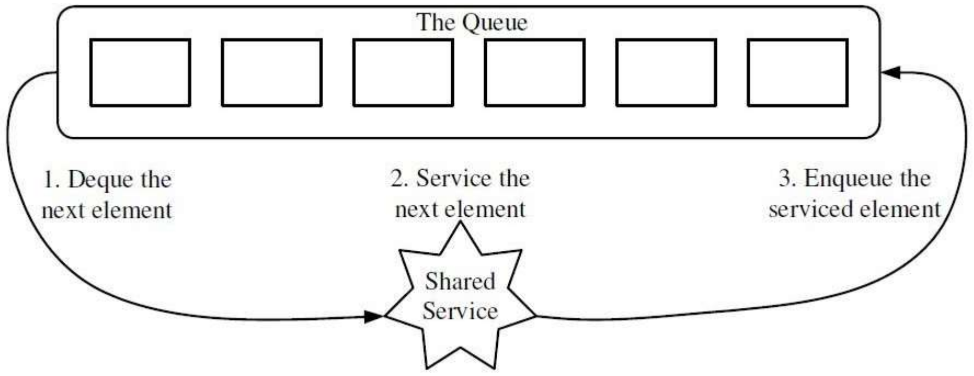
Round-Robin Scheduler

- A ***round-robin*** scheduler iterates through a collection of elements in a circular fashion and "services" each element by performing a given action on it.

- Such a scheduler is used, for example, to fairly allocate a resource that must be shared by a collection of clients.
  - Round-robin scheduling is often used to allocate slices of CPU time to various applications running concurrently on a computer.

- A round-robin scheduler could be implemented with the general queue ADT, by repeatedly performing the following steps on queue Q:

  - (1) **`e = Q.dequeue()`**
  - (2) Service element e
  - (3) **`Q.enqueue(e)`**


The Queue
1. Deque the next element
2. Service the next element
3. Enqueue the serviced element
Shared Service

If **`LinkedQueue`** is used to implement this, there is an increased overhead of removing a node from front and adding one in the end.

With **`CircularQueue`** class, we can implement round-robin scheduler with the following two steps:

- Service element **`Q.front()`**
- **`Q.rotate()`**

Implementing a Queue with a Circularly Linked List

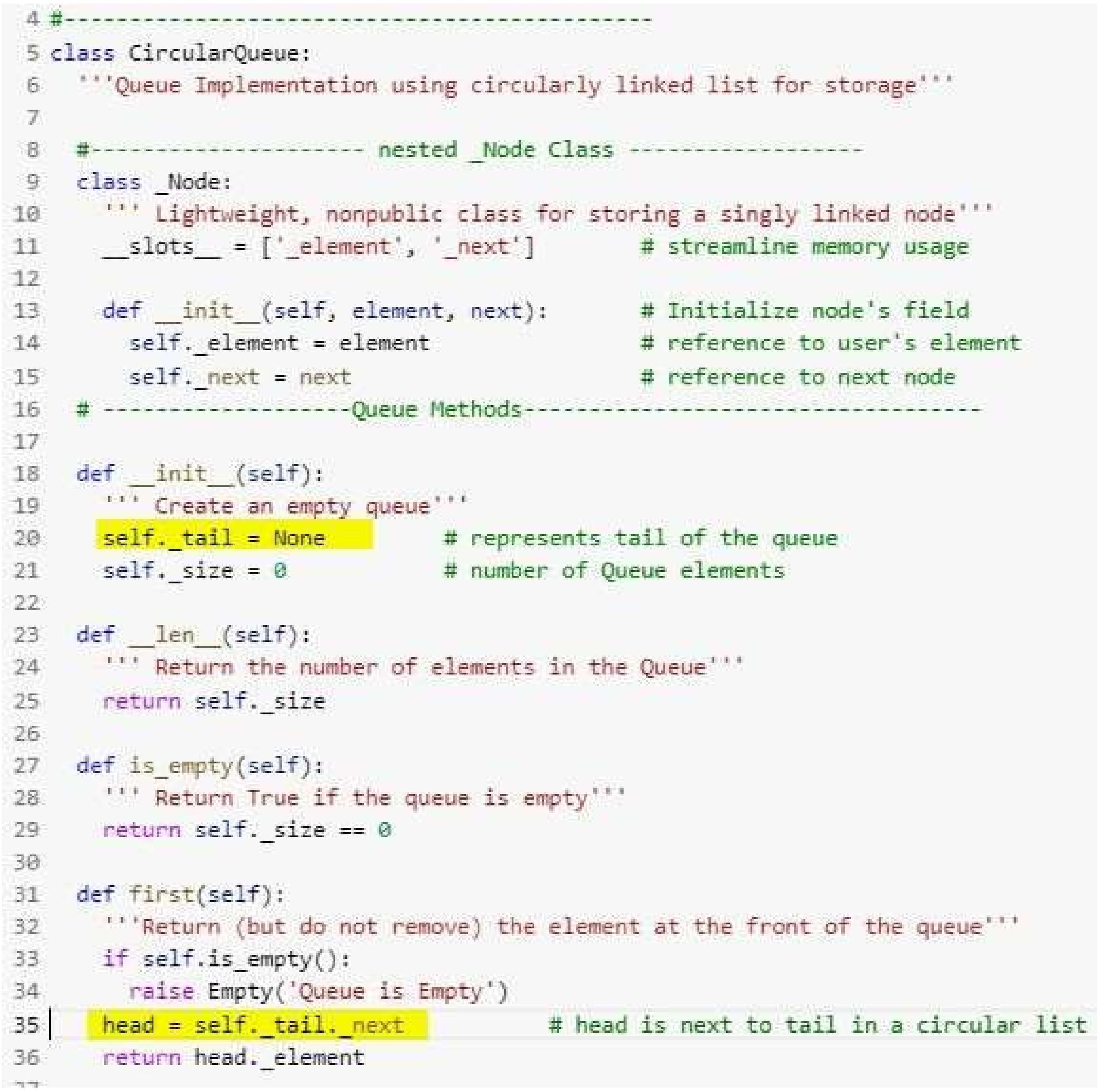
- The implementation maintains only two variables: **`_tail`** and **`_size`**.

- The next reference of the tail is linked to the head.

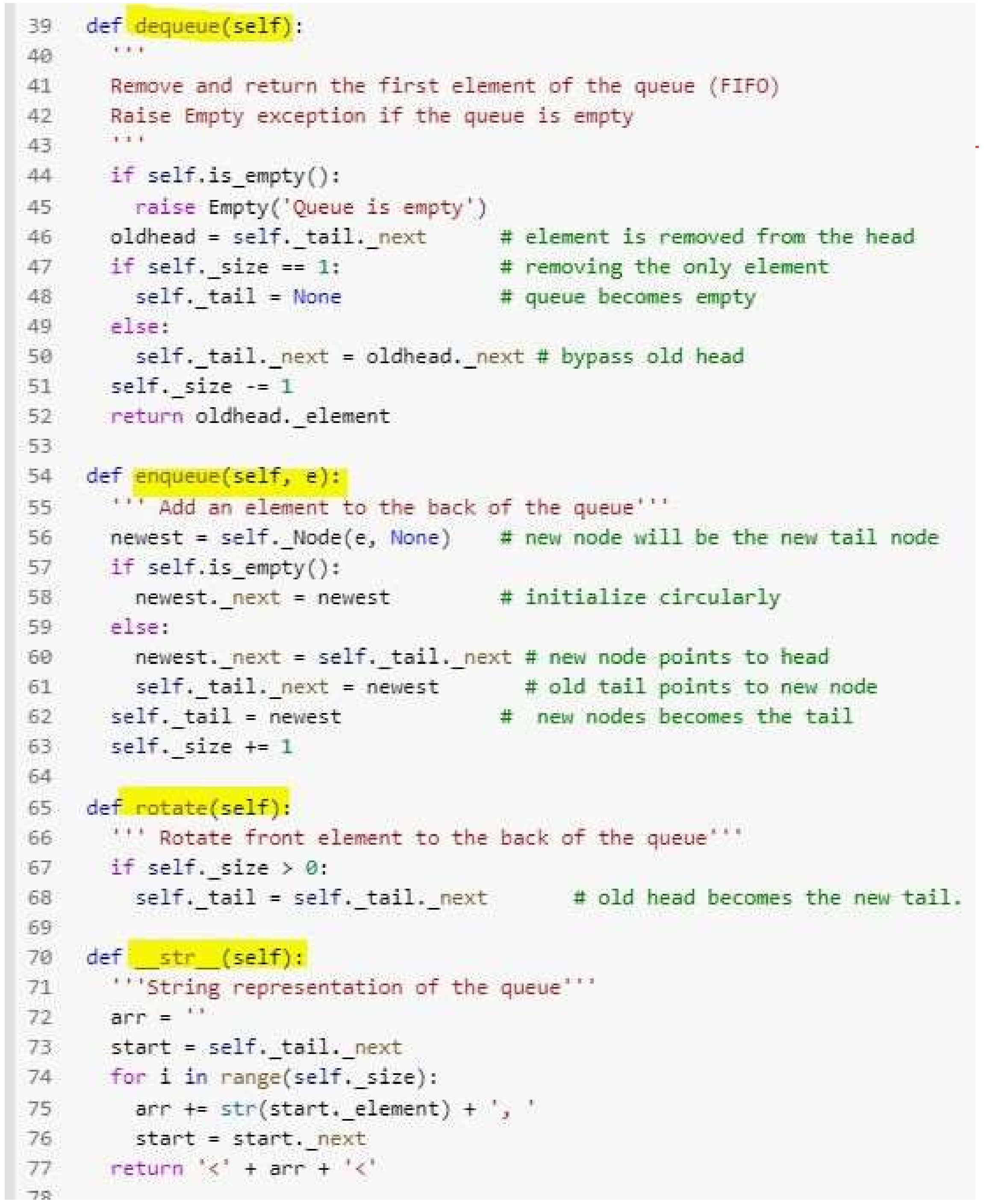
  `head = self._tail._next`

- **`CircularQueue`** class supports an additional 'rotate' method that removes an element from the front and inserts it at the back of the queue. This is achieved by setting the old head as the new tail.

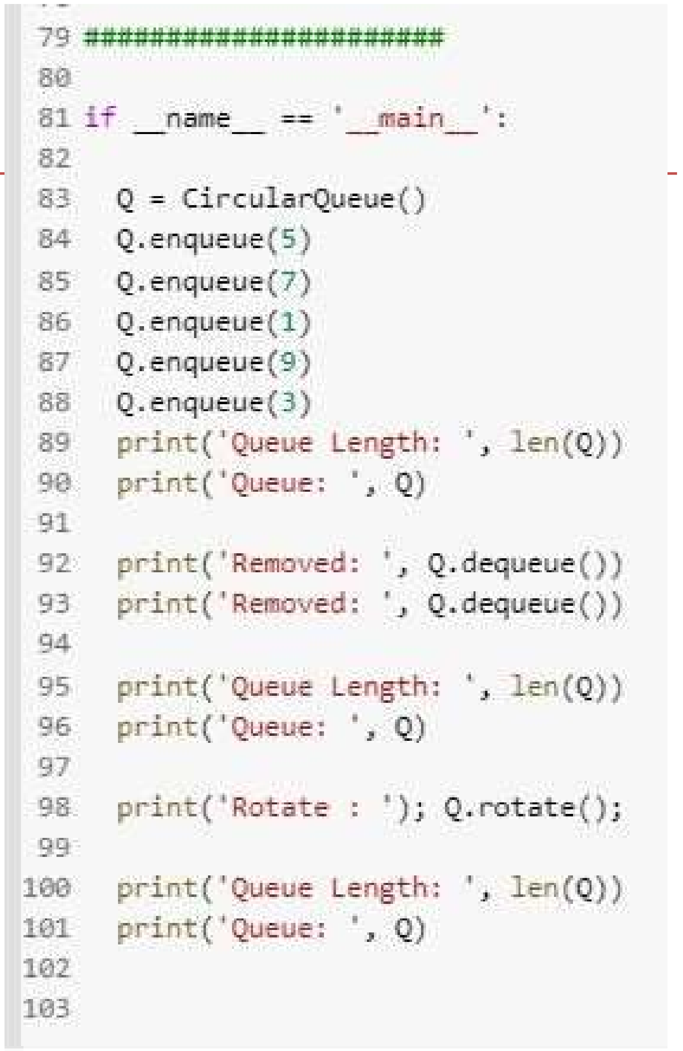
  `self._tail = self._tail._next`



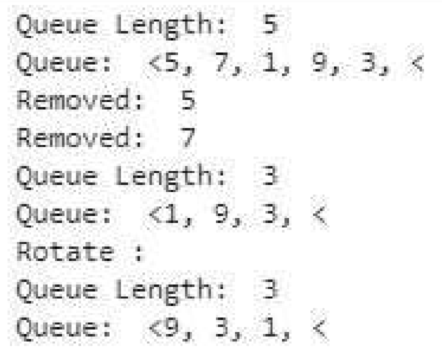
```python
#------------------------------------------------
class CircularQueue:
  '''Queue Implementation using circularly linked list for storage'''

  #-------------------- nested _Node Class -----------------
  class _Node:
    ''' Lightweight, nonpublic class for storing a singly linked node'''
    __slots__ = ['_element', '_next']         # streamline memory usage

    def __init__(self, element, next):        # Initialize node's field
      self._element = element                 # reference to user's element
      self._next = next                       # reference to next node
  # ------------------Queue Methods-----------------------------------

  def __init__(self):
    ''' Create an empty queue'''
    self._tail = None             # represents tail of the queue
    self._size = 0                # number of Queue elements

  def __len__(self):
    ''' Return the number of elements in the Queue'''
    return self._size

  def is_empty(self):
    ''' Return True if the queue is empty'''
    return self._size == 0

  def first(self):
    '''Return (but do not remove) the element at the front of the queue'''
    if self.is_empty():
      raise Empty('Queue is Empty')
    head = self._tail._next              # head is next to tail in a circular list
    return head._element
```



```python
def dequeue(self):
  '''
  Remove and return the first element of the queue (FIFO)
  Raise Empty exception if the queue is empty
  '''
  if self.is_empty():
    raise Empty('Queue is empty')
  oldhead = self._tail._next      # element is removed from the head
  if self._size == 1:             # removing the only element
    self._tail = None             # queue becomes empty
  else:
    self._tail._next = oldhead._next # bypass old head
  self._size -= 1
  return oldhead._element

def enqueue(self, e):
  ''' Add an element to the back of the queue'''
  newest = self._Node(e, None)    # new node will be the new tail node
  if self.is_empty():
    newest._next = newest         # initialize circularly
  else:
    newest._next = self._tail._next # new node points to head
    self._tail._next = newest       # old tail points to new node
  self._tail = newest             #  new nodes becomes the tail
  self._size += 1

def rotate(self):
  ''' Rotate front element to the back of the queue'''
  if self._size > 0:
    self._tail = self._tail._next       # old head becomes the new tail.

def __str__(self):
  '''String representation of the queue'''
  arr = ''
  start = self._tail._next
  for i in range(self._size):
    arr += str(start._element) + ', '
    start = start._next
  return '<' + arr + '<'
```

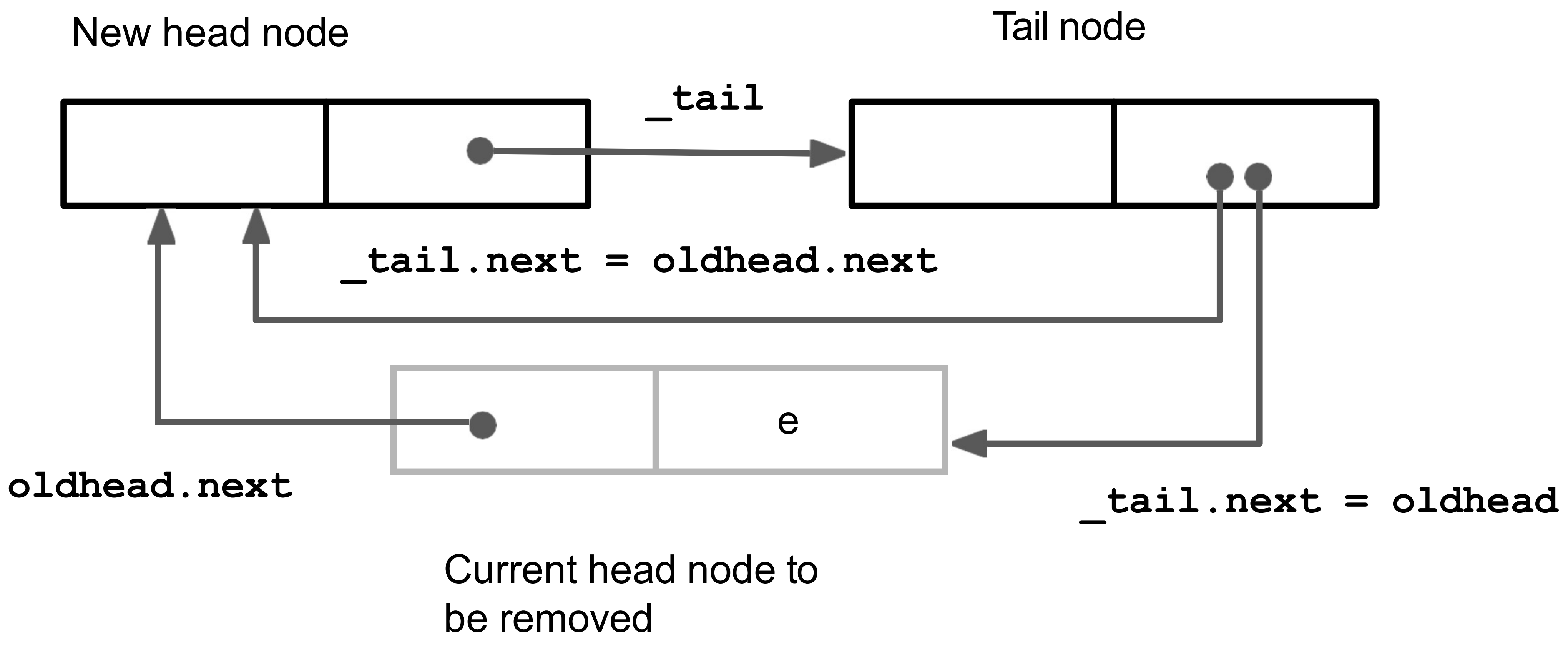


```python
######################

if __name__ == '__main__':

  Q = CircularQueue()
  Q.enqueue(5)
  Q.enqueue(7)
  Q.enqueue(1)
  Q.enqueue(9)
  Q.enqueue(3)
  print('Queue Length: ', len(Q))
  print('Queue: ', Q)

  print('Removed: ', Q.dequeue())
  print('Removed: ', Q.dequeue())

  print('Queue Length: ', len(Q))
  print('Queue: ', Q)

  print('Rotate : '); Q.rotate();

  print('Queue Length: ', len(Q))
  print('Queue: ', Q)
```

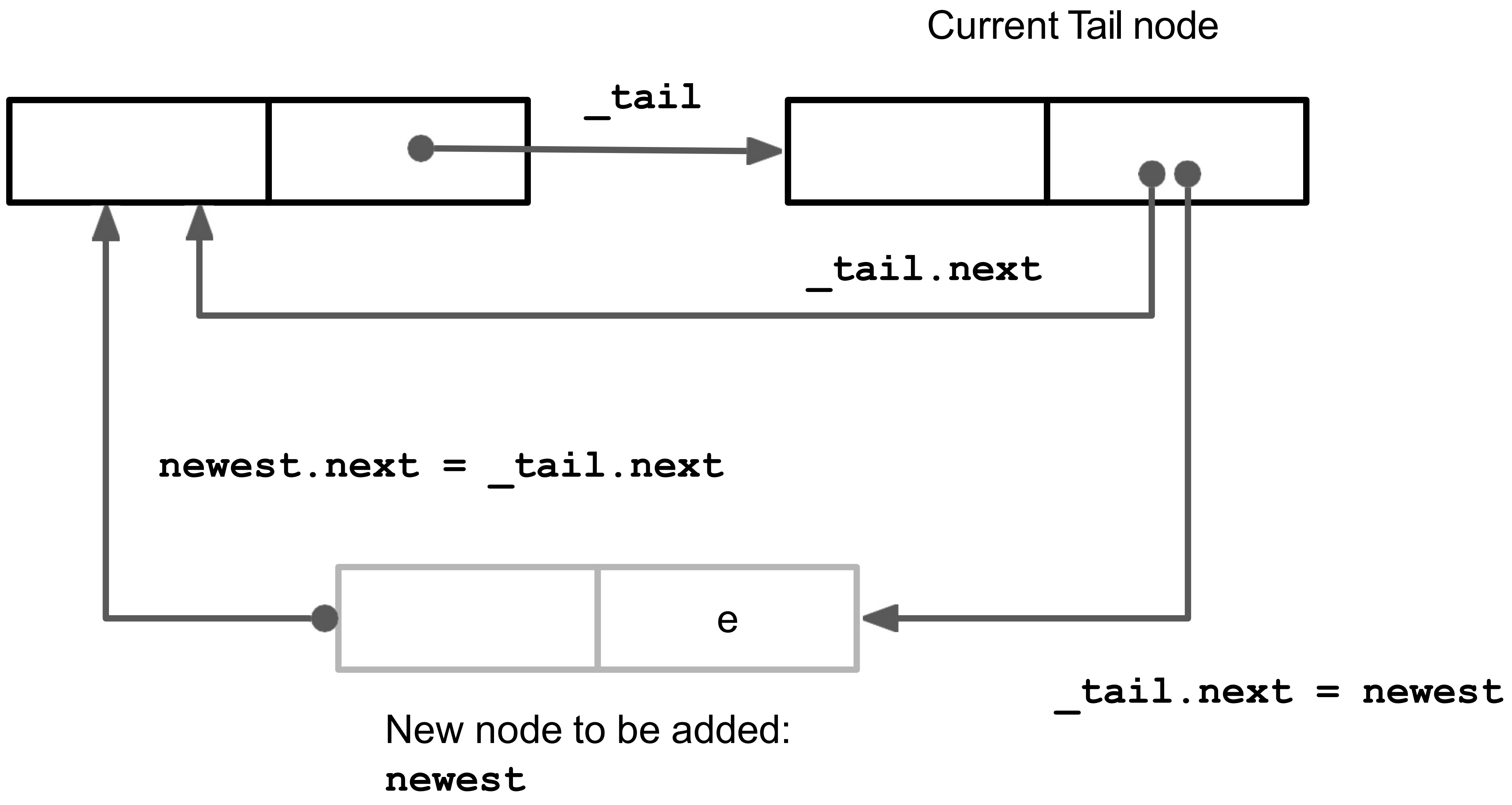


```
Queue Length:  5
Queue:  <5, 7, 1, 9, 3, <
Removed:  5
Removed:  7
Queue Length:  3
Queue:  <1, 9, 3, <
Rotate :
Queue Length:  3
Queue:  <9, 3, 1, <
```

`dequeue()` : Removing element from the beginning of the circular queue (head of the list)

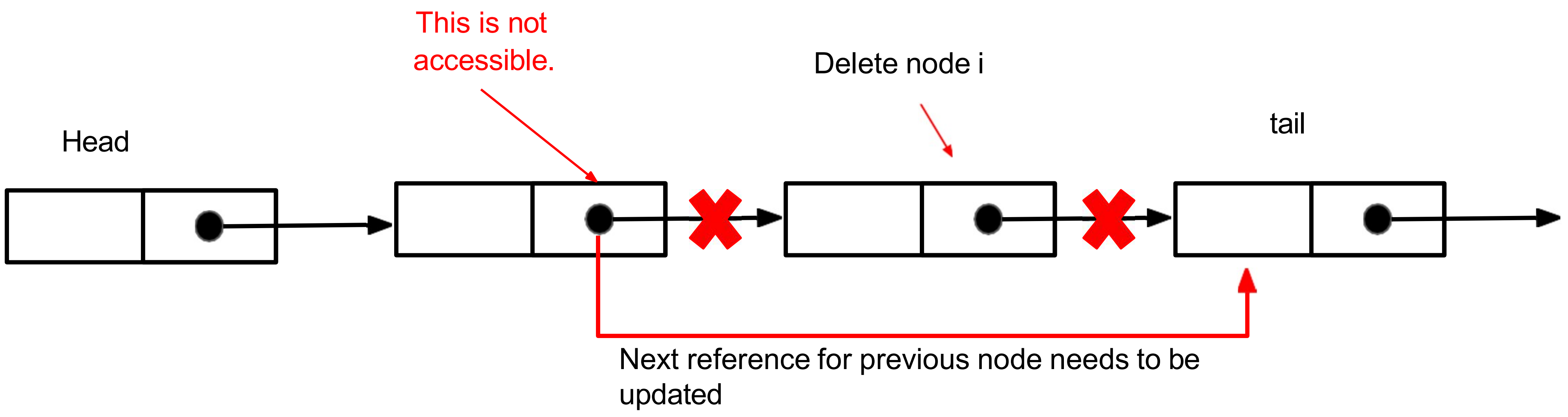


enqueue() : Adding element to the end (tail) of the list



Doubly Linked List

- Motivation:
  - In a singly linked list, each node maintains a reference to the node that is immediately after it.
  - Limitations of Singly Linked List
    - *It is difficult to efficiently delete a node at the tail of the list.*
    - More generally, *we can not efficiently delete an arbitrary node* from an interior position of the list if only given reference to that node - This is because, we can not determine the node that immediately *precedes* the node to be deleted (yet, that node needs to have its next reference updated).



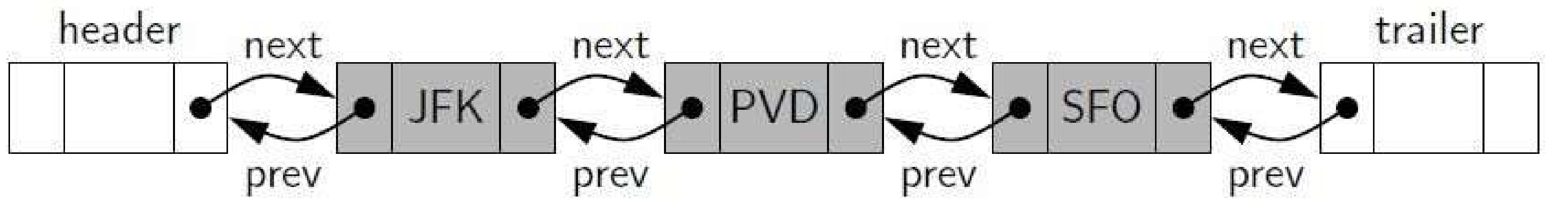
- This limitation of singly linked-list is overcome by a doubly linked list structure where _each node keeps an explicit reference to the node before it and a reference to the node after it._

- This allows _O(1)-time update operations both for insertions and deletions at arbitrary positions_ within the list.

- In order to avoid some special cases when operating near the boundaries of a doubly linked list, it helps to add special nodes at both ends of the list: a ***header*** node at the beginning of the list, and a **trailer** node at the end of the list.

- These dummy nodes (called ***sentinels***) do not store elements.

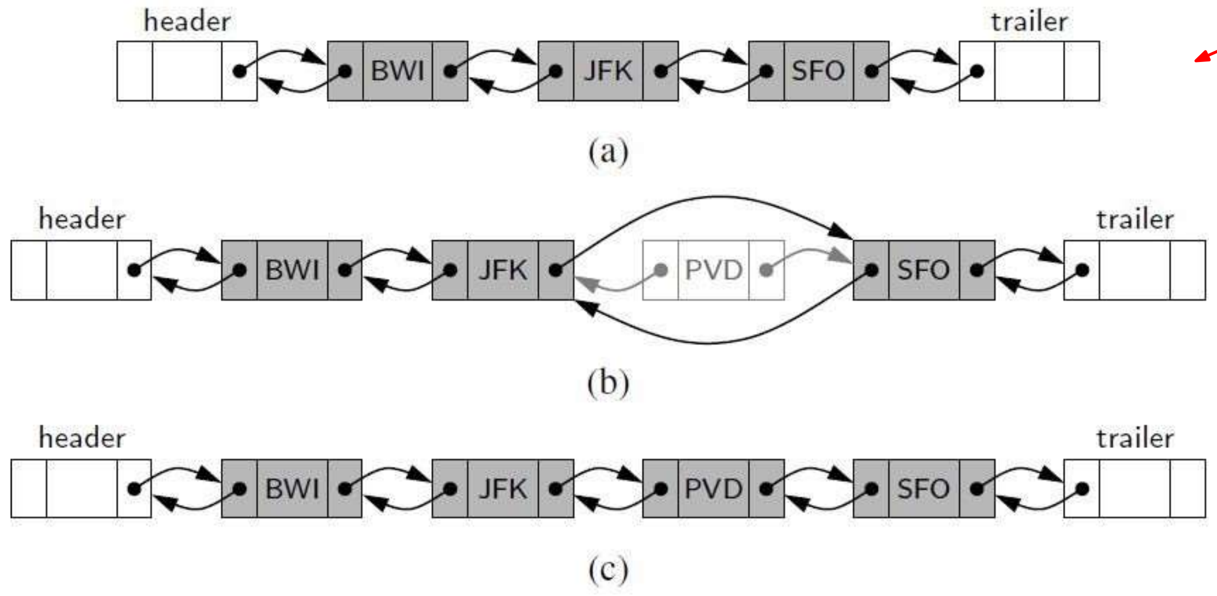
- For an empty list, the next field of the header points to the trailer, and the prev field of the trailer points to the header.

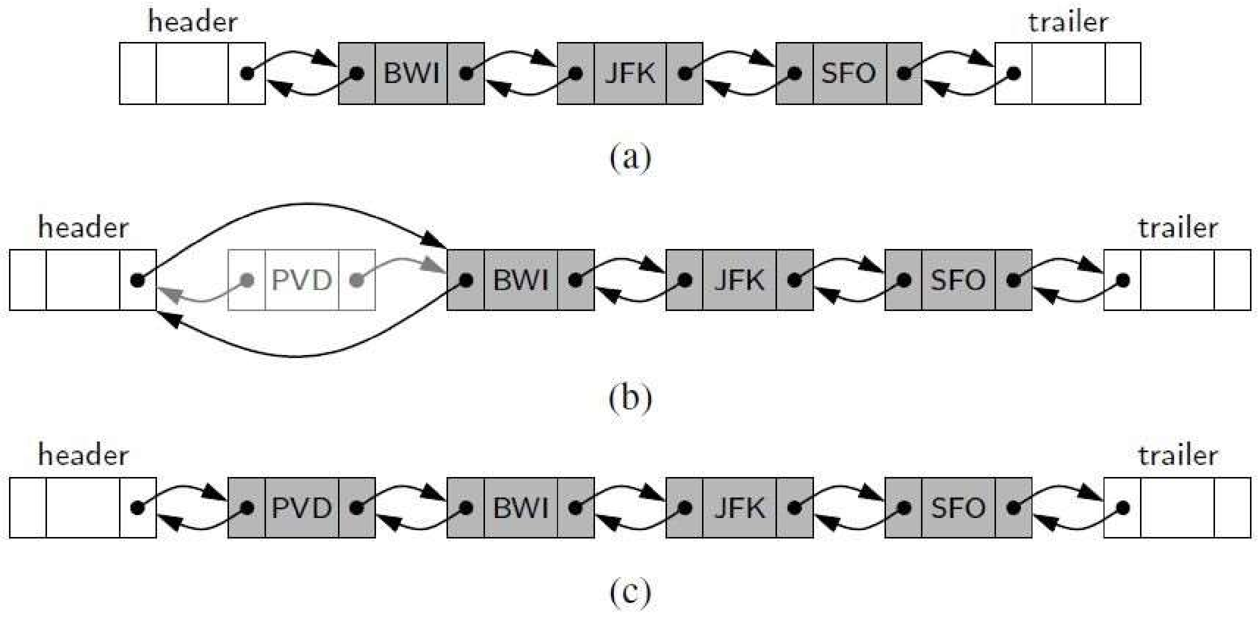


- The use of sentinels simplifies the logic of operations.
- The header and trailer nodes never change—only the nodes between them change.
- All insertions and deletions can be treated in a unified manner - A new node is being inserted or deleted between a pair of existing nodes.

Inserting with a Doubly Linked List

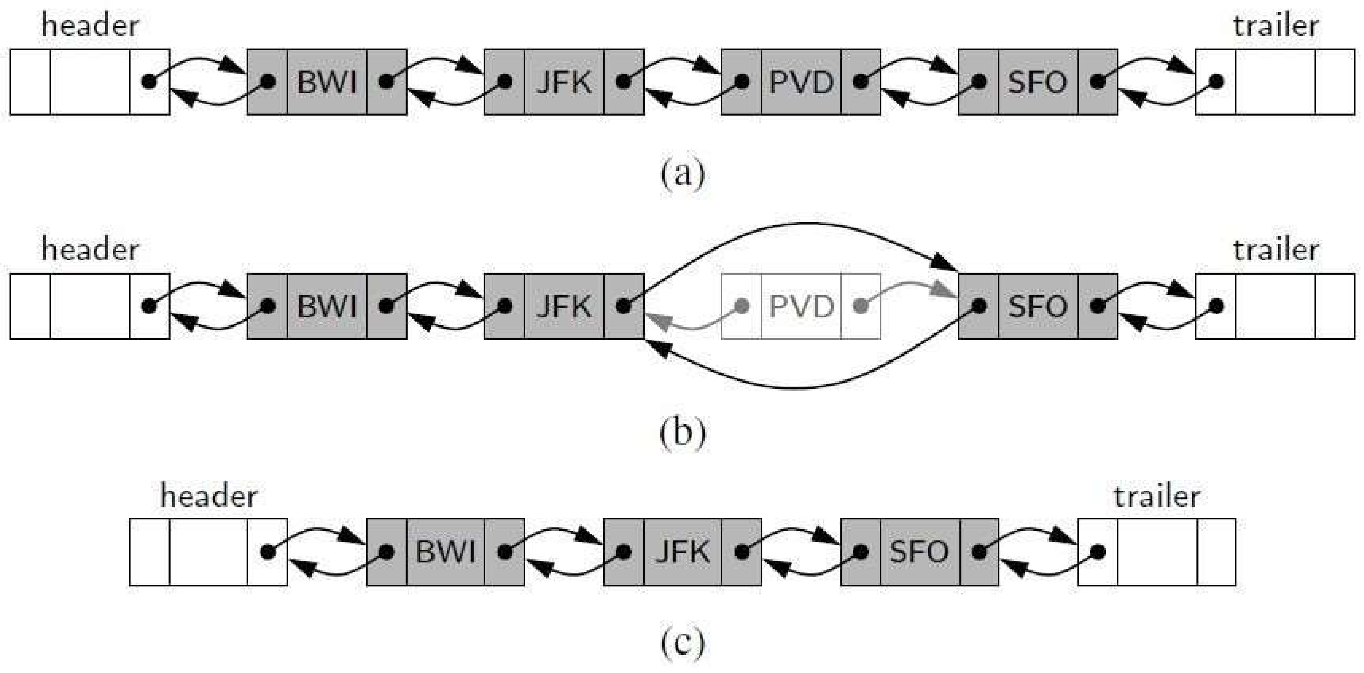
Case I: Inserting at an arbitrary position in a non-empty list.





Case II: Inserting at the beginning of the list

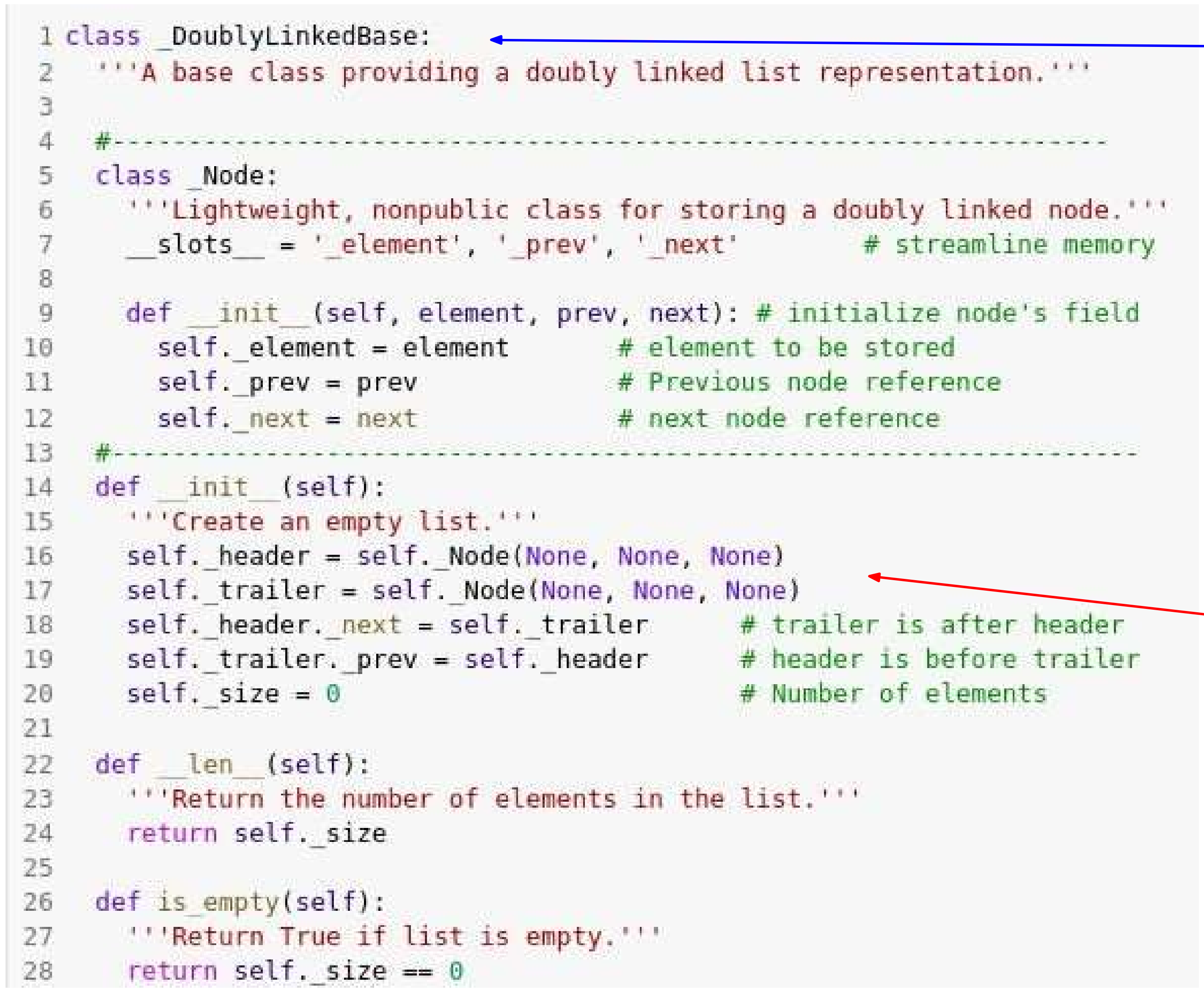
Deleting a Node in a Doubly Linked List



Due to the use of sentinels, same implementation can be used for deleting the first or the last node.

Removing a node at an arbitrary location

Basic Implementation of a Doubly Linked List

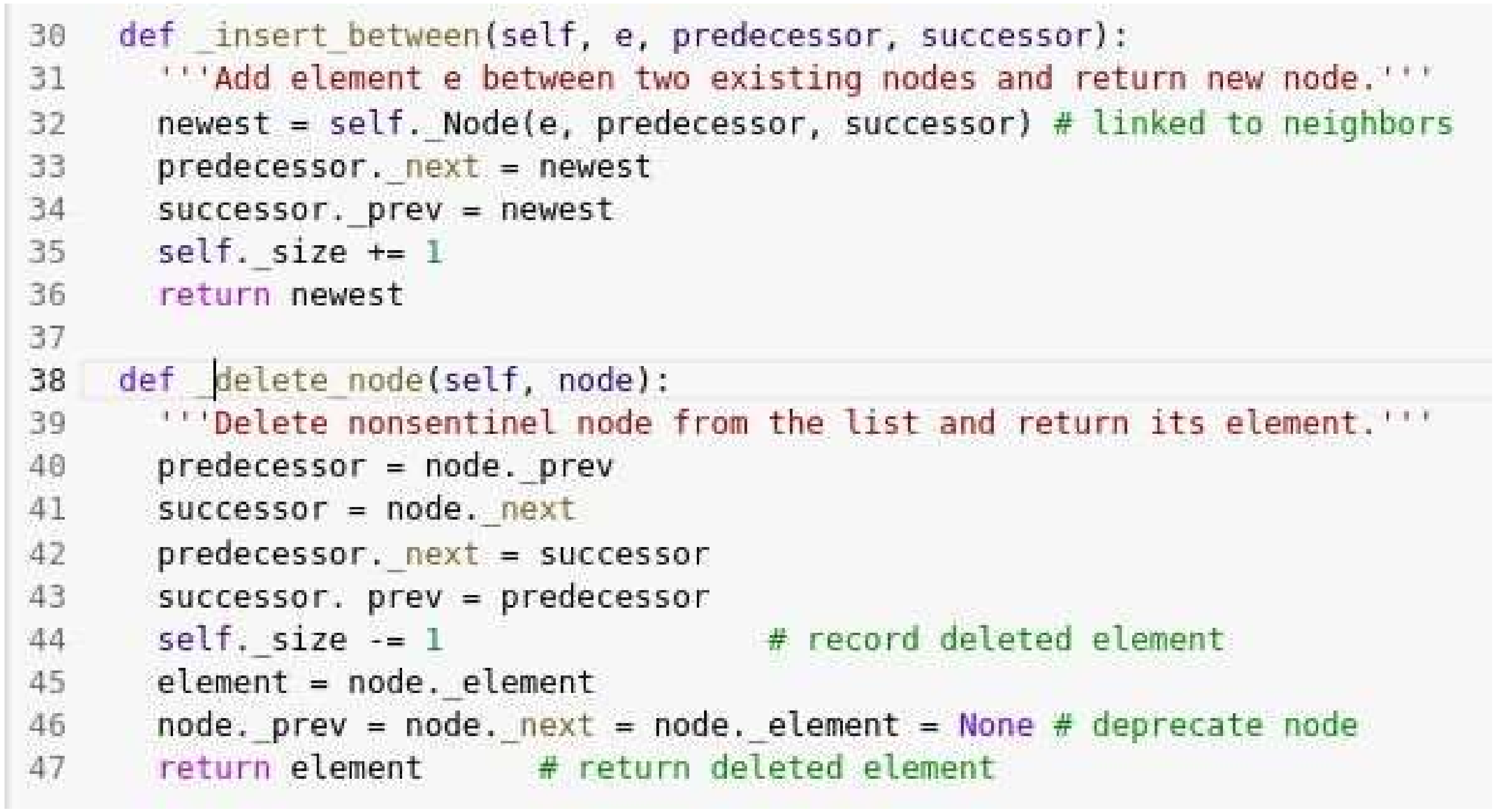


```python
class _DoublyLinkedBase:
  '''A base class providing a doubly linked list representation.'''

  #-------------------------------------------------------------
  class _Node:
    '''Lightweight, nonpublic class for storing a doubly linked node.'''
    __slots__ = '_element', '_prev', '_next'         # streamline memory

    def __init__(self, element, prev, next): # initialize node's field
      self._element = element        # element to be stored
      self._prev = prev              # Previous node reference
      self._next = next              # next node reference
  #---------------------------------------------------------------
  def __init__(self):
    '''Create an empty list.'''
    self._header = self._Node(None, None, None)
    self._trailer = self._Node(None, None, None)
    self._header._next = self._trailer      # trailer is after header
    self._trailer._prev = self._header      # header is before trailer
    self._size = 0                          # Number of elements

  def __len__(self):
    '''Return the number of elements in the list.'''
    return self._size

  def is_empty(self):
    '''Return True if list is empty.'''
    return self._size == 0
```

Base Class declaration

We maintain two references: **`_prev`** & **`_next`**

Performs insertion and deletion of nodes as per the algorithm described earlier (see Slides 24 & 25)



```
def _insert_between(self, e, predecessor, successor):
  '''Add element e between two existing nodes and return new node.'''
  newest = self._Node(e, predecessor, successor) # linked to neighbors
  predecessor._next = newest
  successor._prev = newest
  self._size += 1
  return newest

def _delete_node(self, node):
  '''Delete nonsentinel node from the list and return its element.'''
  predecessor = node._prev
  successor = node._next
  predecessor._next = successor
  successor._prev = predecessor
  self._size -= 1                 # record deleted element
  element = node._element
  node._prev = node._next = node._element = None # deprecate node
  return element      # return deleted element
```

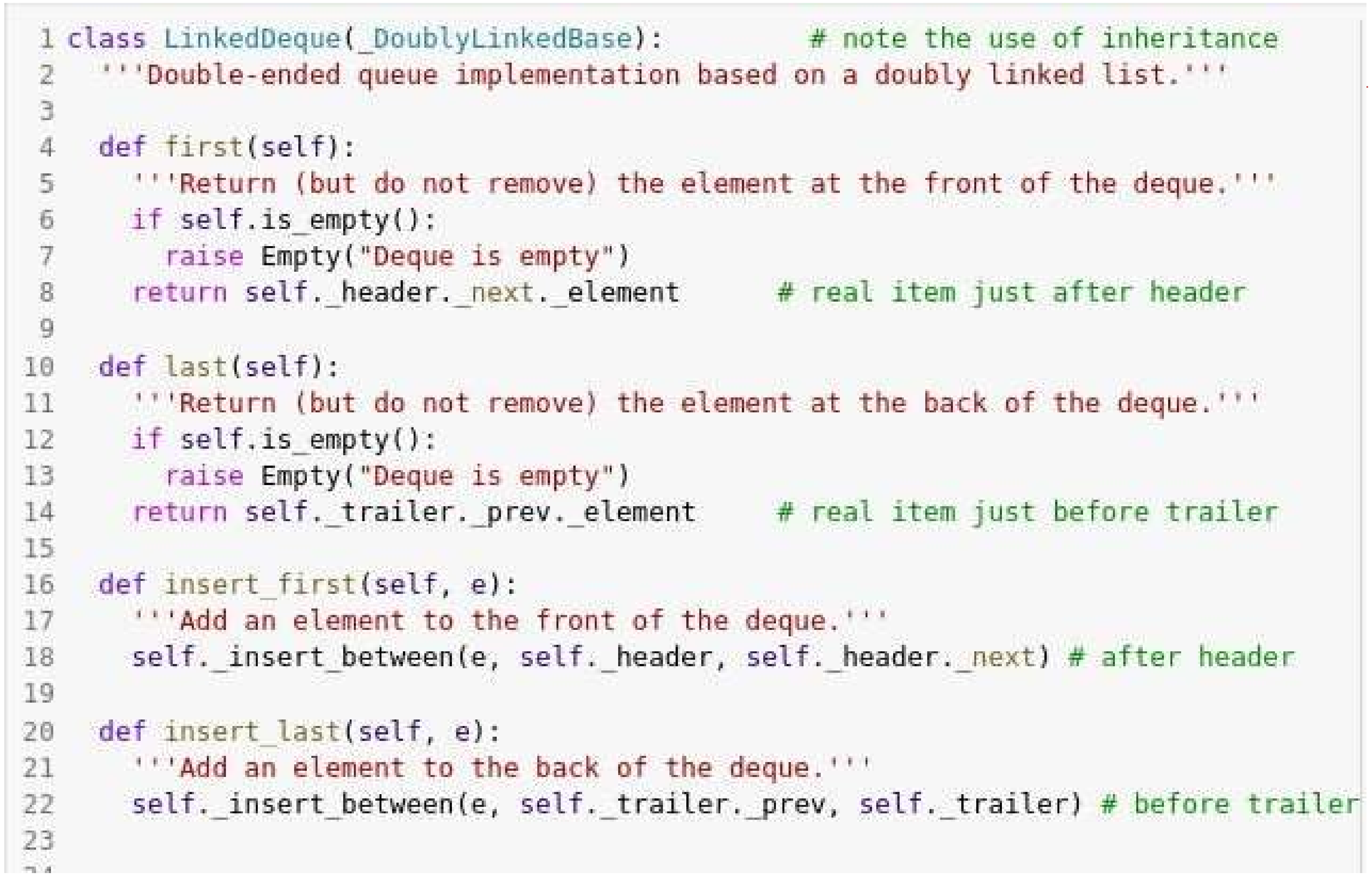
Implementing Deque with Doubly Linked List

- Previous array-based deque (double-end queues) implementation achieved operations in ***amortized*** O(1) time, due to occasional need to resize the array.

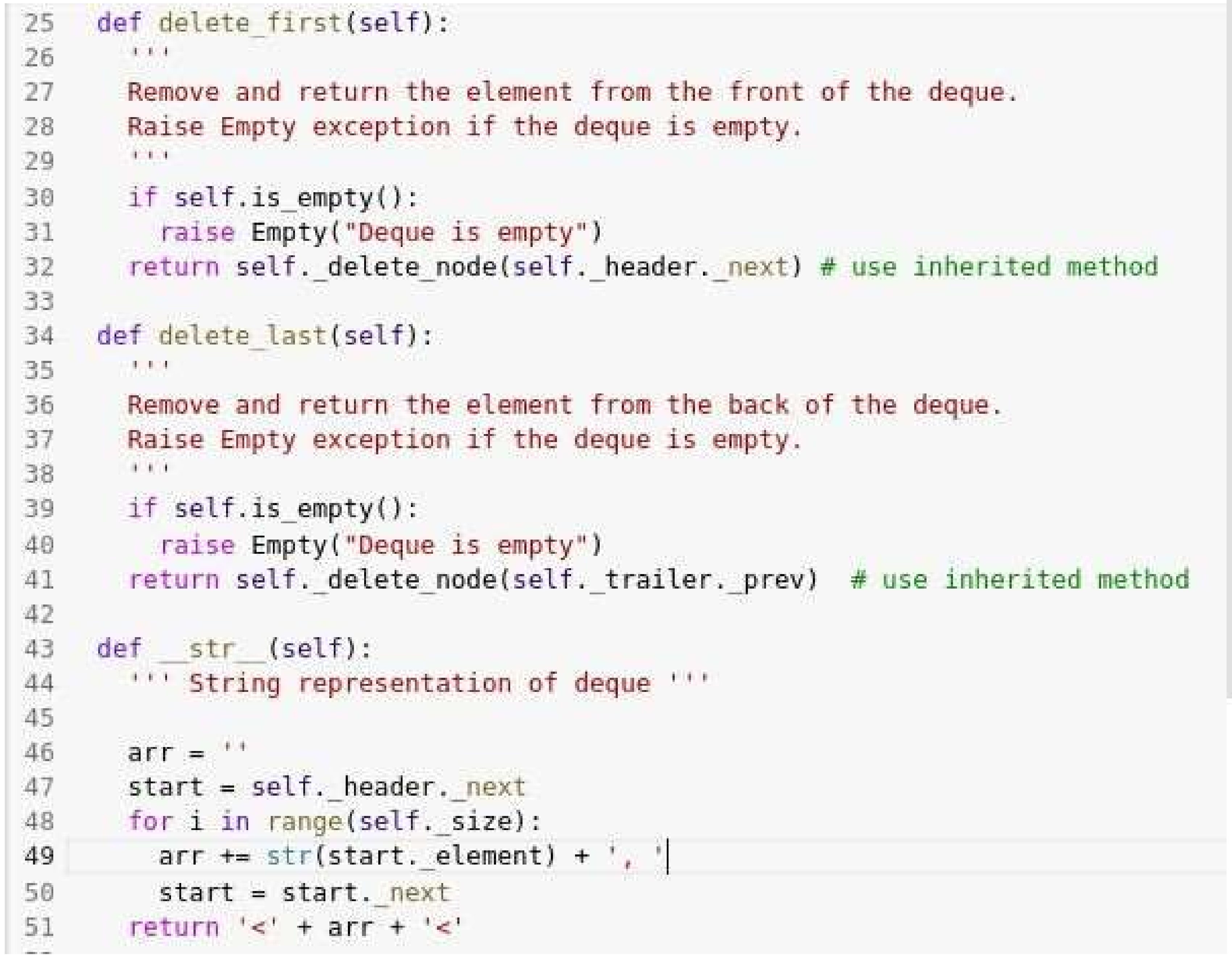
- With an implementation based upon a doubly linked list, we can achieve all deque operation in ***worst-case*** O(1) time.

- With the use of sentinels, take note of the following changes:
    - the header does not store the first element of the deque—it is the node just after the header that stores the first element (assuming the deque is nonempty).
    - Similarly, the node just before the trailer stores the last element of the deque.

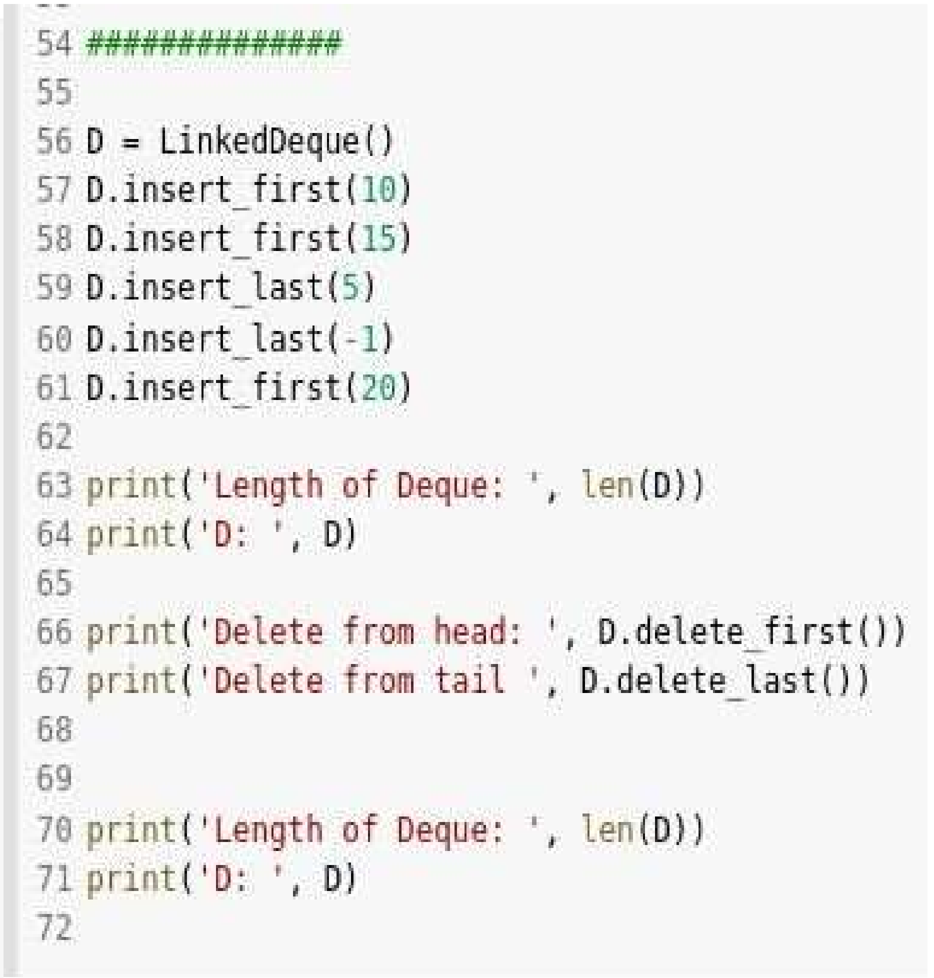
- **`_insert_between()`** and **`_delete_node()`** methods are inherited in the child class to implement insert or delete nodes at the beginning or the end of deque.



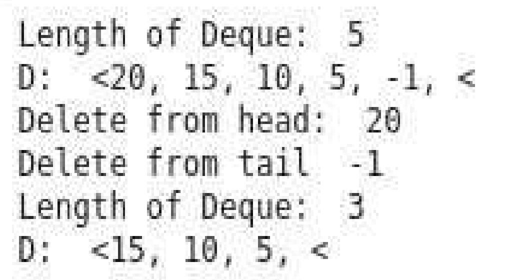
```python
class LinkedDeque(_DoublyLinkedBase):         # note the use of inheritance
  '''Double-ended queue implementation based on a doubly linked list.'''

  def first(self):
    '''Return (but do not remove) the element at the front of the deque.'''
    if self.is_empty():
      raise Empty("Deque is empty")
    return self._header._next._element      # real item just after header

  def last(self):
    '''Return (but do not remove) the element at the back of the deque.'''
    if self.is_empty():
      raise Empty("Deque is empty")
    return self._trailer._prev._element     # real item just before trailer

  def insert_first(self, e):
    '''Add an element to the front of the deque.'''
    self._insert_between(e, self._header, self._header._next) # after header

  def insert_last(self, e):
    '''Add an element to the back of the deque.'''
    self._insert_between(e, self._trailer._prev, self._trailer) # before trailer

```



```python
  def delete_first(self):
    '''
    Remove and return the element from the front of the deque.
    Raise Empty exception if the deque is empty.
    '''
    if self.is_empty():
      raise Empty("Deque is empty")
    return self._delete_node(self._header._next) # use inherited method

  def delete_last(self):
    '''
    Remove and return the element from the back of the deque.
    Raise Empty exception if the deque is empty.
    '''
    if self.is_empty():
      raise Empty("Deque is empty")
    return self._delete_node(self._trailer._prev)  # use inherited method

  def __str__(self):
    ''' String representation of deque '''

    arr = ''
    start = self._header._next
    for i in range(self._size):
      arr += str(start._element) + ', '
      start = start._next
    return '<' + arr + '<'
```



```python
##############

D = LinkedDeque()
D.insert_first(10)
D.insert_first(15)
D.insert_last(5)
D.insert_last(-1)
D.insert_first(20)

print('Length of Deque: ', len(D))
print('D: ', D)

print('Delete from head: ', D.delete_first())
print('Delete from tail ', D.delete_last())


print('Length of Deque: ', len(D))
print('D: ', D)
```



```
Length of Deque:  5
D:  <20, 15, 10, 5, -1, <
Delete from head:  20
Delete from tail  -1
Length of Deque:  3
D:  <15, 10, 5, <
```

The Positional List ADT

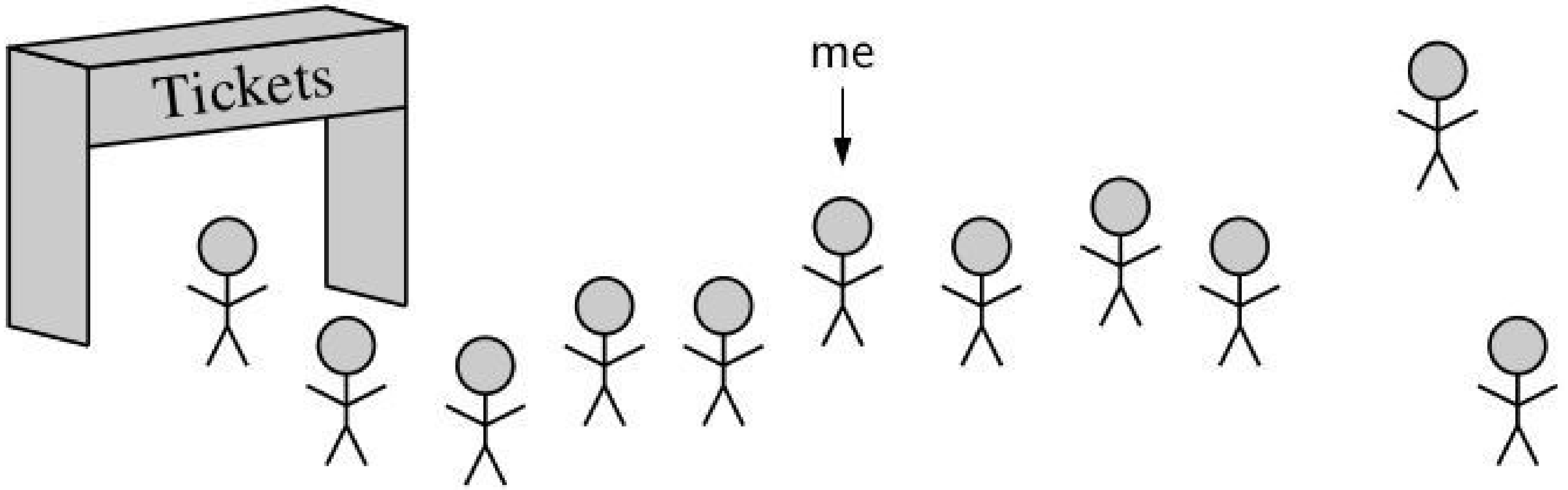
- The abstract data types that we have considered thus far, namely stacks, queues, and double-ended queues, only allow update operations that occur at one end of a sequence or the other.

- Consider the previous queue example for customer waiting for receiving a service.
    - What if a customer wants to leave the queue before reaching the ticket counter?
    - What if someone who is waiting in line to buy tickets allows a friend to “cut” into line at that position.

- This motivates having a more generalized ADT that provides a user a way to refer to elements anywhere in a sequence, and to perform arbitrary insertions and deletions.

- In array-based sequences, integer indices provide an excellent means for describing the location of an element, or the location at which an insertion or deletion should take place.

- However, numeric indices are not a good choice for describing positions within a linked list
  - We cannot efficiently access an entry knowing only its index;
    - finding an element at a given index within a linked list requires traversing the list incrementally from its beginning or end, counting elements as we go.
  - Indices are not a good abstraction for describing a local position in some applications
    - because the index of an entry changes over time due to insertions or deletions that happen earlier in the sequence.
    - For example, it may not be convenient to describe the location of a person waiting in line by knowing precisely how far away that person is from the front of the line.



We wish to be able to identify the position of an element in a sequence without the use of an integer index.

Another Example: Word Processor

- A text document can be viewed as a long sequence of characters.

- A word processor uses the abstraction of a ***cursor*** to describe a position within the document without explicit use of an integer index, allowing operations such as “delete the character at the cursor” or “insert a new character just after the cursor.”

- Furthermore, we may be able to refer to an inherent position within a document, such as the beginning of a particular section, without relying on a character index (or even a section number) that may change as the document evolves.

Can we use a Node reference as a Position?

- Linked-lists provide O(1)-time insertion and deletions at arbitrary positions as long as a reference to a relevant node is available.

- It is therefore very tempting to develop an ADT in which a node reference serves as the mechanism for describing a position.

- In fact, our **`_DoublyLinkedBase`** class has methods **`_insert_between()`** and **`_delete_node()`** that accept node references as parameters.

- However, *such direct use of nodes would violate the object-oriented design principles of abstraction* and encapsulation that were introduced in an earlier lecture.

- There are several reasons for encapsulating the nodes of a linked list:
    - It will be simpler for users if they need not bother about the details of implementation such as, low level manipulation of node references or our reliance of the use of sentinel nodes.
    - We can ensure robustness of data structure by not permitting users to directly access or manipulate node references by preventing mismanagement by users.
    - Encapsulation provides flexibility to redesign data structure and improves its performance.

The Positional List ADT

- Provides a general abstraction of a sequence of elements with the ability to identify the location of an element.

- Also provides a ***simpler*** *position abstract data type* to describe a location within a list.

  - A position acts as a marker or token within the broader positional list.
  - A position p is unaffected by changes elsewhere in a list unless it is deleted by issuing an explicit command.

- A position instance is a simple object, supporting only the following method:
  `p.element()`: Return the element stored at position p.

- Positions serve as parameters to some methods and as return values from other methods.

The Positional List ADT (L) supports the following ***accessor*** methods:

- **`L.first()`**: Return the position of the first element of L, or None if L is empty.
- **`L.last()`**: Return the position of the last element of L, or None if L is empty.
- **`L.before(p)`**: Return the position of L immediately before position p, or None if p is the first position.
- **`L.after(p)`**: Return the position of L immediately after position p, or None if p is the last position.
- **`L.is_empty()`**: Return True if list L does not contain any elements.
- **`len(L)`**: Return the number of elements in the list.
- **`iter(L)`**: Return a forward iterator for the elements of the list.

The positional list ADT also includes the following update methods:

- `L.add_first(e)`: Insert a new element e at the front of L, returning the position of the new element.
- `L.add_last(e)`: Insert a new element e at the back of L, returning the position of the new element.
- `L.add_before(p,e)`: Insert a new element e just before position p in L, returning the position of the new element.
- `L.add_after(p,e)`: Insert a new element e just after position p in L, returning the position of the new element.
- `L.replace(p,e)`: Replace the element at position p with element e, returning the element formerly at position p.
- `L.delete(p)`: Remove and return the element at position p in L, invalidating the position.

Few Observations:

- Note that the **`first()`** and **`last()`** methods of the positional list ADT return the associated positions, not the elements.

- The element at a position can be obtained by invoking element method on that position.

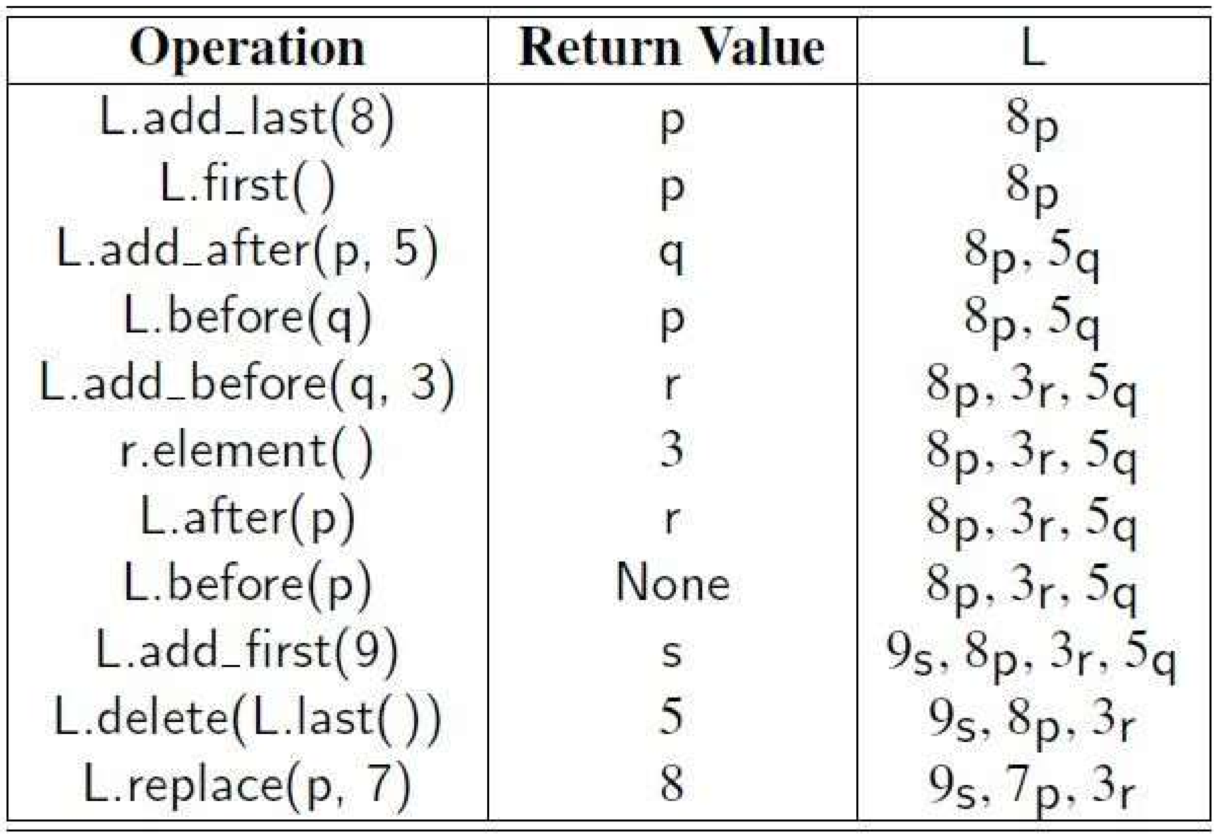
  `L.first().element()`: first element in the list
  `L.last().element()`: last element in the list

- The advantage of receiving a position as a return value is that we can use that position to navigate the list.

```
cursor = data.first()
while cursor is not None:
  print(cursor.element())       # print the element stored at the position
  cursor = data.after(cursor)   # advance to the next position (if any)
```

- Because the ADT includes support for Python's **`iter`** function, users may rely on the traditional for-loop syntax for such a forward traversal of a list named data.

```
for e in data:
  print(e)
```



Operation	Return Value	L
L.add_last(8)	p	8_p
L.first()	p	8_p
L.add_after(p, 5)	q	$8_p, 5_q$
L.before(q)	p	$8_p, 5_q$
L.add_before(q, 3)	r	$8_p, 3_r, 5_q$
r.element()	3	$8_p, 3_r, 5_q$
L.after(p)	r	$8_p, 3_r, 5_q$
L.before(p)	None	$8_p, 3_r, 5_q$
L.add_first(9)	s	$9_s, 8_p, 3_r, 5_q$
L.delete(L.last())	5	$9_s, 8_p, 3_r$
L.replace(p, 7)	8	$9_s, 7_p, 3_r$

Series of Operations on an initially empty positional list L.

Positional List Implementation using Doubly Linked List

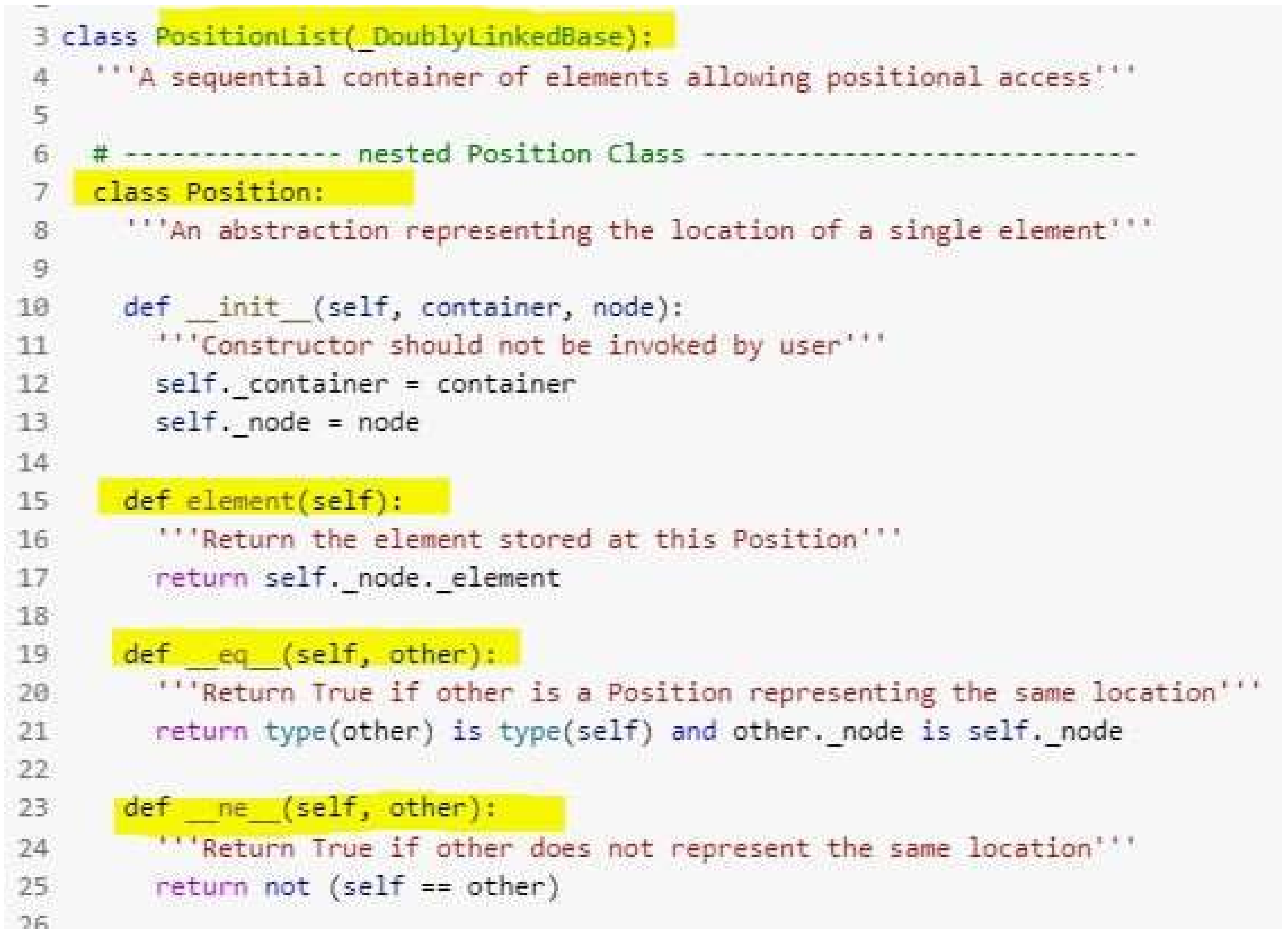
- ***Proposition:*** Each method of the positional list ADT runs in worst-case O(1) time when implemented with a doubly linked list.

- We rely on the **`_DoublyLinkedBase`** class for our low-level representation.

- Provide a public interface in accordance with the positional list ADT.

- We define a **`Position`** Class nested within the **`PositionList`** Class.

- Our various **`PositionalList`** methods may end up creating redundant **`Position`** instances that reference the same underlying node. Therefore, we define **`__eq__`** and **`__ne__`** special methods within the Position class to carry out tests such as **`p == q`**.



```python
class PositionList(_DoublyLinkedBase):
  '''A sequential container of elements allowing positional access'''

  # -------------- nested Position Class ---------------------------
  class Position:
    '''An abstraction representing the location of a single element'''

    def __init__(self, container, node):
      '''Constructor should not be invoked by user'''
      self._container = container
      self._node = node

    def element(self):
      '''Return the element stored at this Position'''
      return self._node._element

    def __eq__(self, other):
      '''Return True if other is a Position representing the same location'''
      return type(other) is type(self) and other._node is self._node

    def __ne__(self, other):
      '''Return True if other does not represent the same location'''
      return not (self == other)
```

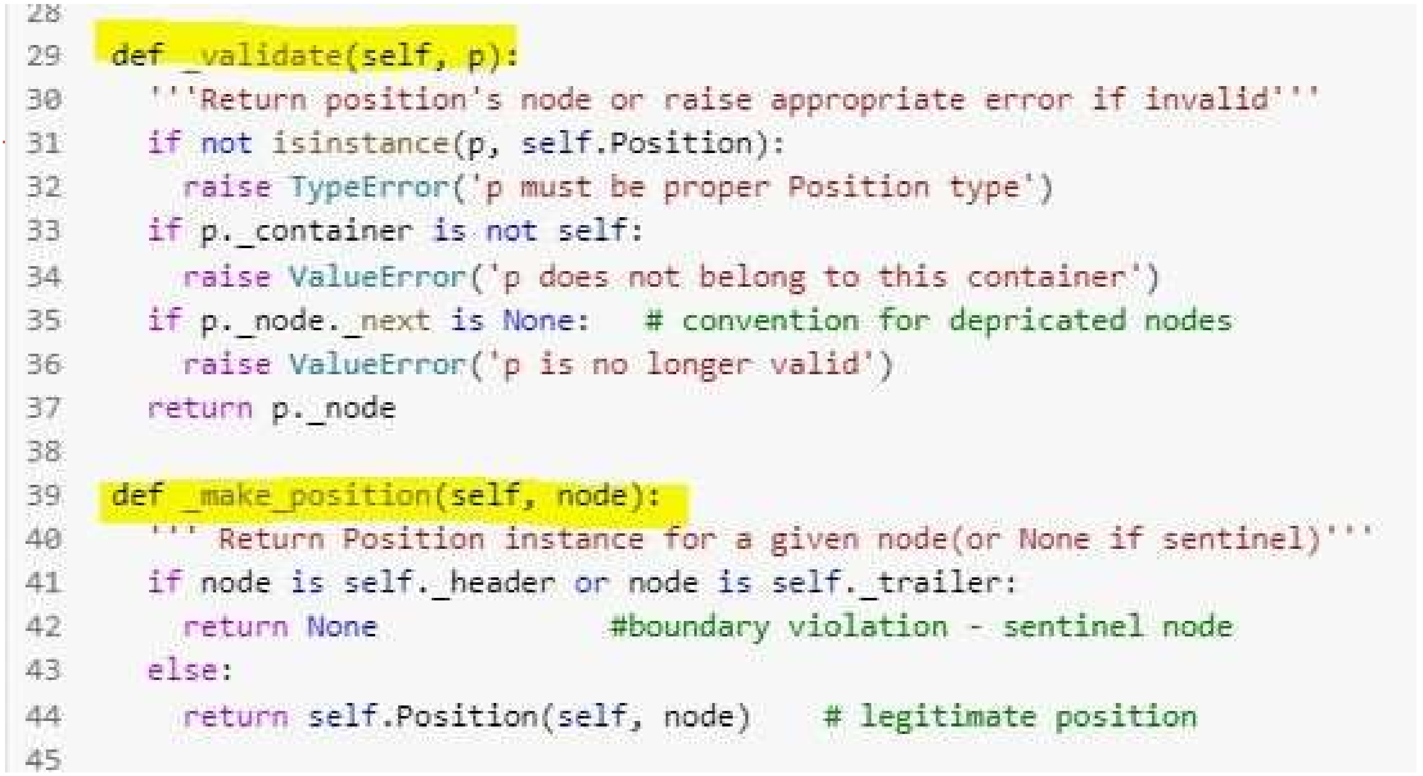
Validating Positions:

- Each time a method of the **`PositionalList`** class accepts a position as a parameter, we want to verify that the position is valid, and if so, to determine the underlying node associated with the position.

- This functionality is implemented by a nonpublic method named **`_validate`**.

- Internally, a **`position`** maintains a reference to the associated node of the linked list, and also a reference to the list instance that contains the specified node.

- With the container reference, we can robustly detect when a caller sends a position instance that does not belong to the indicated list.



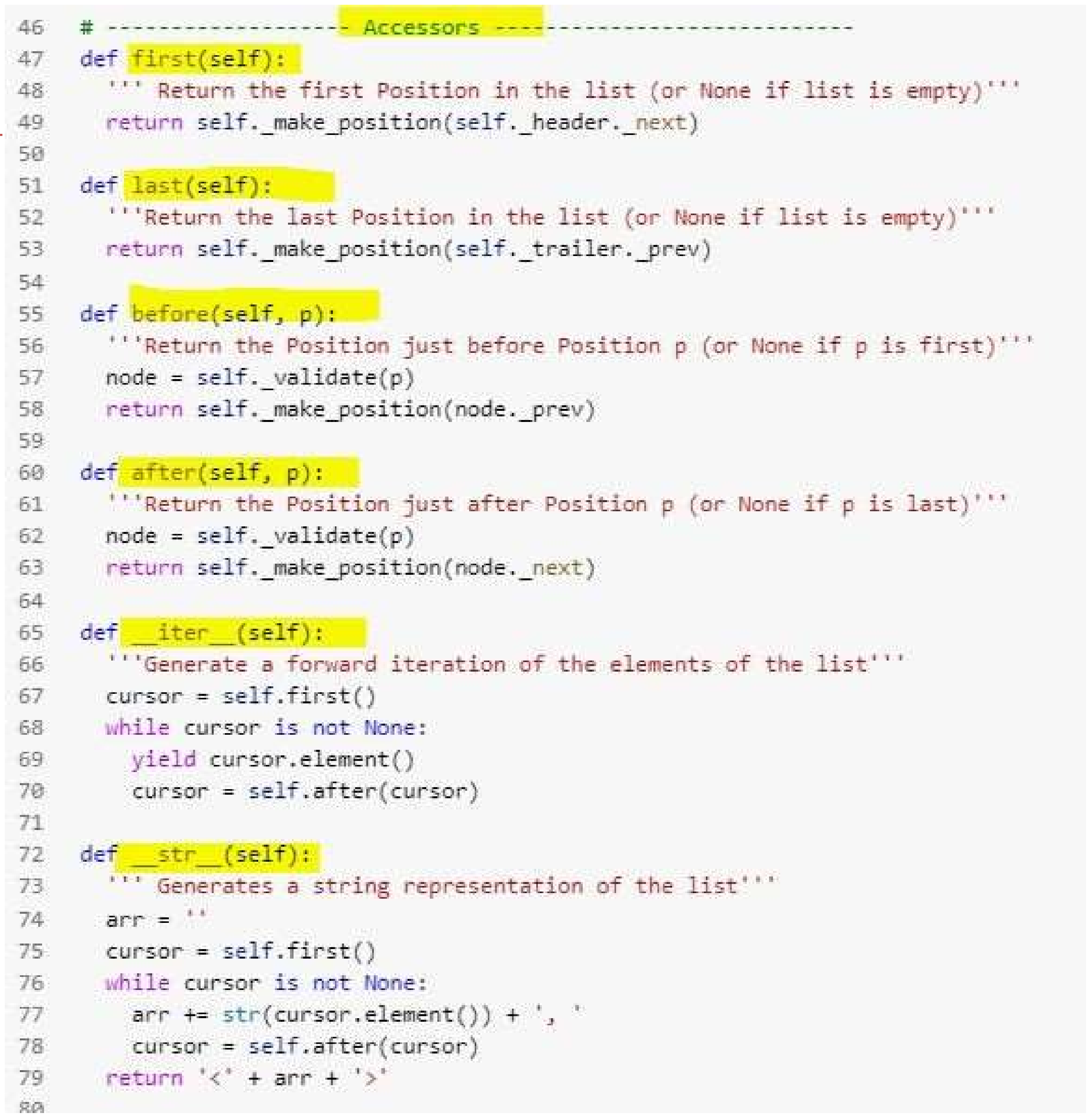
```
def _validate(self, p):
  '''Return position's node or raise appropriate error if invalid'''
  if not isinstance(p, self.Position):
    raise TypeError('p must be proper Position type')
  if p._container is not self:
    raise ValueError('p does not belong to this container')
  if p._node._next is None:   # convention for depricated nodes
    raise ValueError('p is no longer valid')
  return p._node

def _make_position(self, node):
  ''' Return Position instance for a given node(or None if sentinel)'''
  if node is self._header or node is self._trailer:
    return None             #boundary violation - sentinel node
  else:
    return self.Position(self, node)    # legitimate position
```

- We can also detect a position instance that belongs to the list, but that refers to a node that is no longer part of that list.

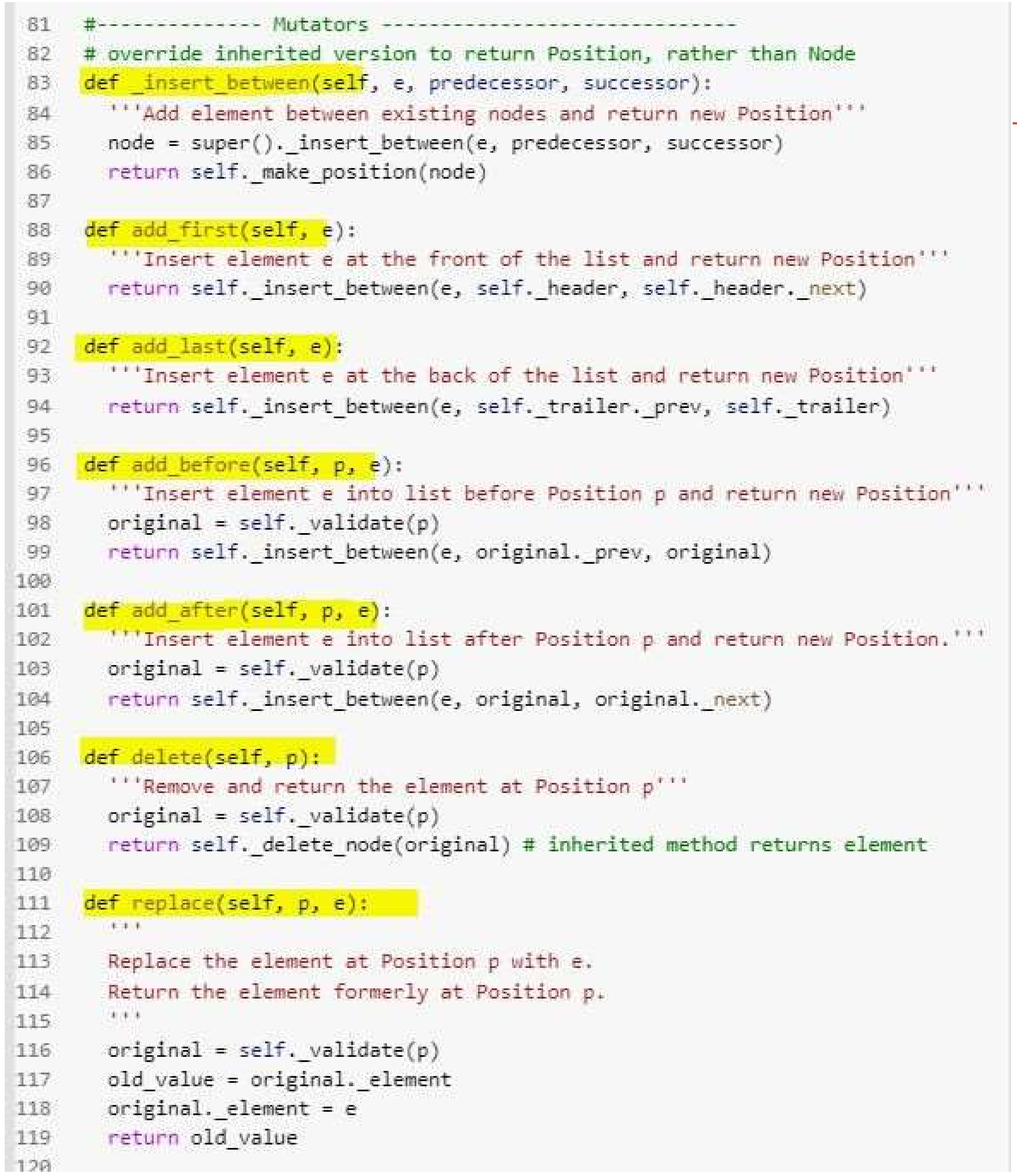
- The **`_delete_node()`** of the base class sets the previous and next references of a deleted node to None; we can recognize that condition to detect a deprecated node.

Access and Update Methods:

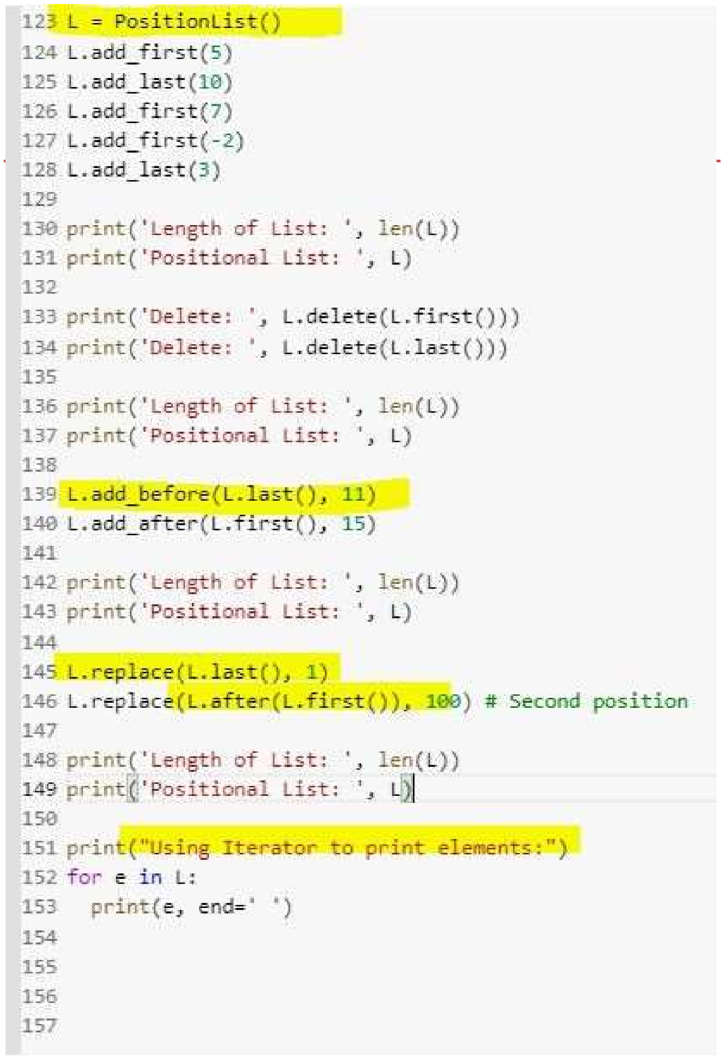
- All of these methods trivially adapt the underlying doubly linked list implementation to support the public interface of the positional list ADT.
- These methods rely on the **`_validate`** utility to "unwrap" any position that is sent.
- They also rely on a **`_make_position`** utility to "wrap" nodes as **`Position`** instances to return to the user, making sure never to return a position referencing a sentinel.
- For convenience, the inherited **`_insert_between`** utility method is overridden so that ours returns a ***position*** associated with the newly created node (whereas the inherited version returns the node itself).



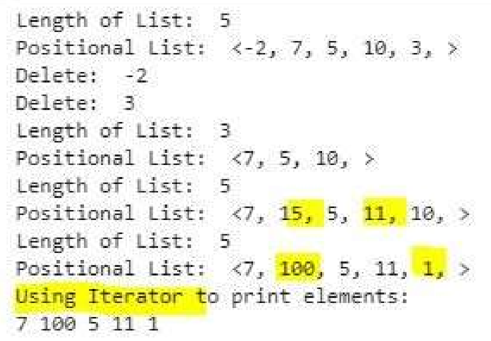
```
# ------------------- Accessors ---------------------------
def first(self):
  ''' Return the first Position in the list (or None if list is empty)'''
  return self._make_position(self._header._next)

def last(self):
  '''Return the last Position in the list (or None if list is empty)'''
  return self._make_position(self._trailer._prev)

def before(self, p):
  '''Return the Position just before Position p (or None if p is first)'''
  node = self._validate(p)
  return self._make_position(node._prev)

def after(self, p):
  '''Return the Position just after Position p (or None if p is last)'''
  node = self._validate(p)
  return self._make_position(node._next)

def __iter__(self):
  '''Generate a forward iteration of the elements of the list'''
  cursor = self.first()
  while cursor is not None:
    yield cursor.element()
    cursor = self.after(cursor)

def __str__(self):
  ''' Generates a string representation of the list'''
  arr = ''
  cursor = self.first()
  while cursor is not None:
    arr += str(cursor.element()) + ', '
    cursor = self.after(cursor)
  return '<' + arr + '>'
```



```
#-------------- Mutators ------------------------------
# override inherited version to return Position, rather than Node
def _insert_between(self, e, predecessor, successor):
  '''Add element between existing nodes and return new Position'''
  node = super()._insert_between(e, predecessor, successor)
  return self._make_position(node)

def add_first(self, e):
  '''Insert element e at the front of the list and return new Position'''
  return self._insert_between(e, self._header, self._header._next)

def add_last(self, e):
  '''Insert element e at the back of the list and return new Position'''
  return self._insert_between(e, self._trailer._prev, self._trailer)

def add_before(self, p, e):
  '''Insert element e into list before Position p and return new Position'''
  original = self._validate(p)
  return self._insert_between(e, original._prev, original)

def add_after(self, p, e):
  '''Insert element e into list after Position p and return new Position.'''
  original = self._validate(p)
  return self._insert_between(e, original, original._next)

def delete(self, p):
  '''Remove and return the element at Position p'''
  original = self._validate(p)
  return self._delete_node(original) # inherited method returns element

def replace(self, p, e):
  '''
  Replace the element at Position p with e.
  Return the element formerly at Position p.
  '''
  original = self._validate(p)
  old_value = original._element
  original._element = e
  return old_value
```



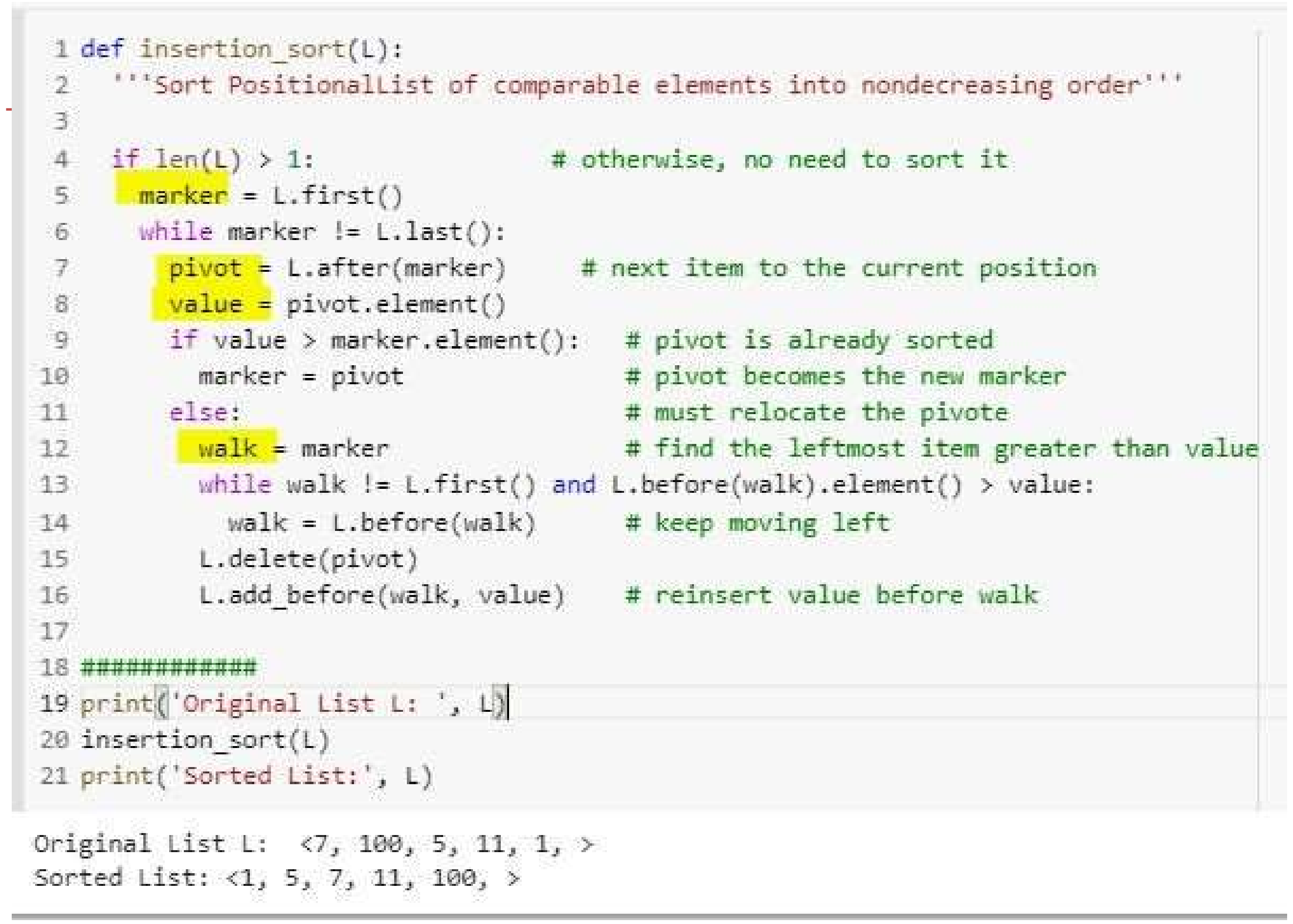
```
L = PositionList()
L.add_first(5)
L.add_last(10)
L.add_first(7)
L.add_first(-2)
L.add_last(3)

print('Length of List: ', len(L))
print('Positional List: ', L)

print('Delete: ', L.delete(L.first()))
print('Delete: ', L.delete(L.last()))

print('Length of List: ', len(L))
print('Positional List: ', L)

L.add_before(L.last(), 11)
L.add_after(L.first(), 15)

print('Length of List: ', len(L))
print('Positional List: ', L)

L.replace(L.last(), 1)
L.replace(L.after(L.first()), 100) # Second position

print('Length of List: ', len(L))
print('Positional List: ', L)

print("Using Iterator to print elements:")
for e in L:
  print(e, end=' ')
```



```
Length of List:  5
Positional List:  <-2, 7, 5, 10, 3, >
Delete:  -2
Delete:  3
Length of List:  3
Positional List:  <7, 5, 10, >
Length of List:  5
Positional List:  <7, 15, 5, 11, 10, >
Length of List:  5
Positional List:  <7, 100, 5, 11, 1, >
Using Iterator to print elements:
7 100 5 11 1
```

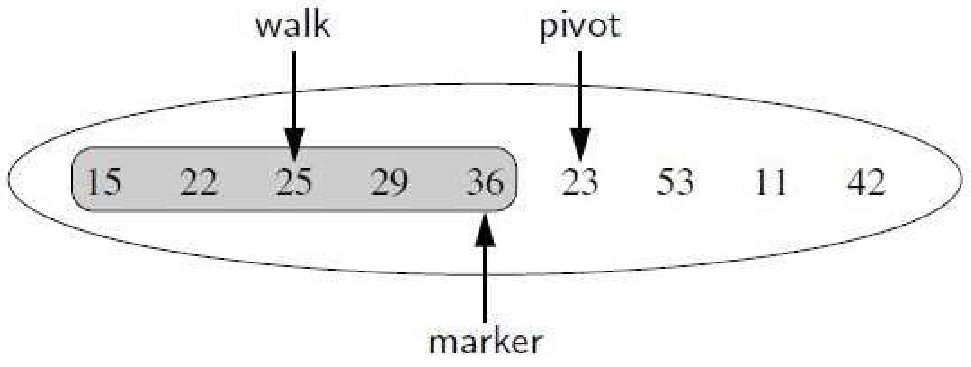
Sorting a Positional List

- We implement insertion-sort algorithm
- We maintain three variables:
  - **Marker** - the rightmost position of the currently sorted portion of a list.
  - **Pivot** - the position just after the marker.
  - **Walk** - to move leftward from the marker, as long as there remains a preceding element with value larger than the pivot's.



```
def insertion_sort(L):
  '''Sort PositionalList of comparable elements into nondecreasing order'''

  if len(L) > 1:                # otherwise, no need to sort it
    marker = L.first()
    while marker != L.last():
      pivot = L.after(marker)       # next item to the current position
      value = pivot.element()
      if value > marker.element():   # pivot is already sorted
        marker = pivot               # pivot becomes the new marker
      else:                          # must relocate the pivote
        walk = marker                # find the leftmost item greater than value
        while walk != L.first() and L.before(walk).element() > value:
          walk = L.before(walk)      # keep moving left
        L.delete(pivot)
        L.add_before(walk, value)    # reinsert value before walk

#############
print('Original List L: ', L)
insertion_sort(L)
print('Sorted List:', L)
```

```
Original List L:  <7, 100, 5, 11, 1, >
Sorted List: <1, 5, 7, 11, 100, >
```



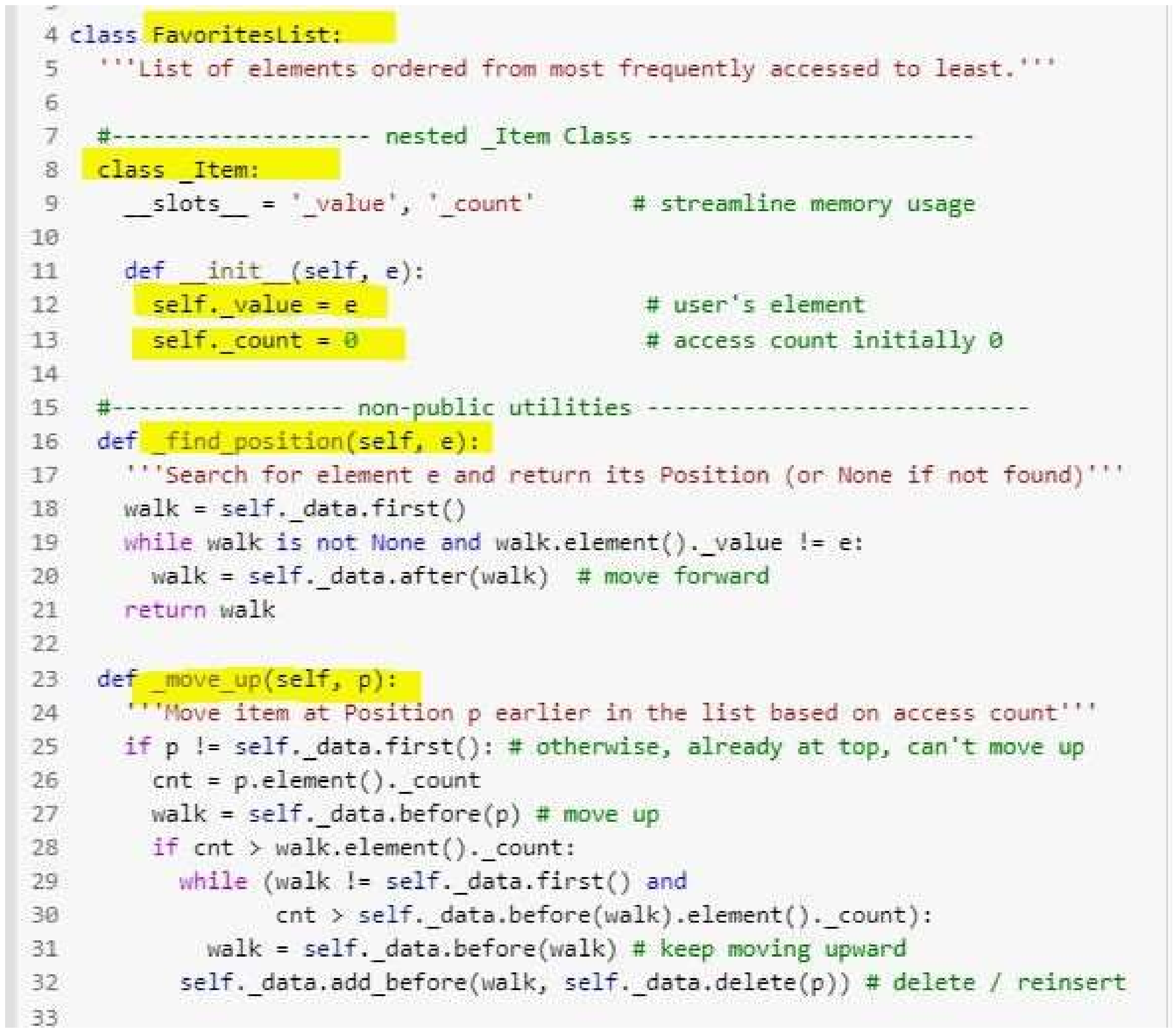
Case Study: Maintaining Access Frequencies

Favorites List ADT

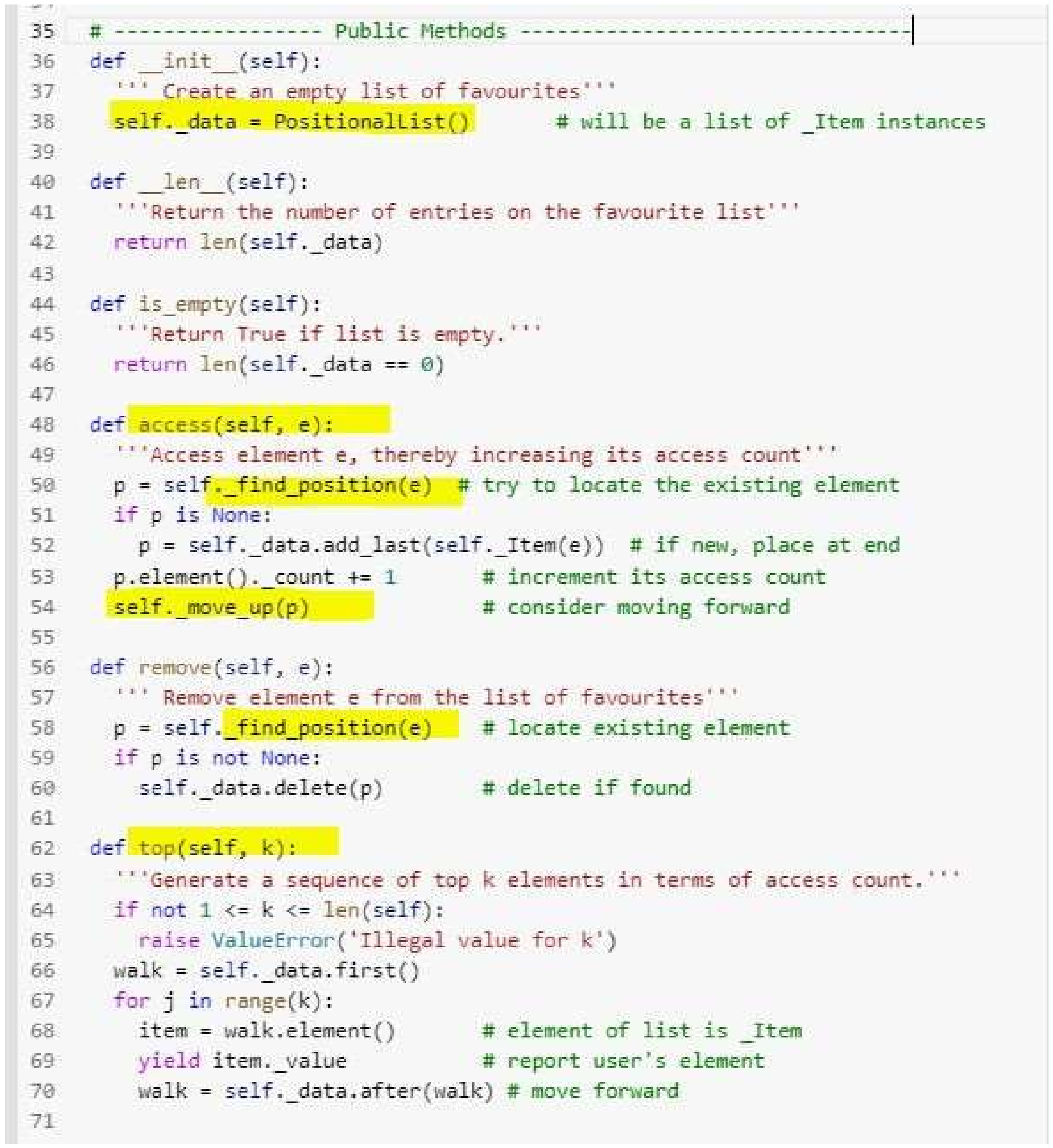
- For maintaining a collection of elements while keeping track of the number of times each element is accessed.
- Examples:
    - Web browser that keeps track of a user’s most accessed URLs.
    - A music collection that maintains a list of the most frequently played songs for a user.
- Favorites List ADT should provide the following methods:
    - **`access(e)`:** Access the element e, incrementing its access count, and adding it to the favorites list if it is not already present.
    - **`remove(e)`:** Remove element e from the favorites list, if present.
    - **`top(k)`:** Return an iteration of the k most accessed elements.

- The favourite list is sorted.
    - Every time an item is accessed, its count is incremented and its position is adjusted within the list using a technique similar to insertion-sort algorithm.

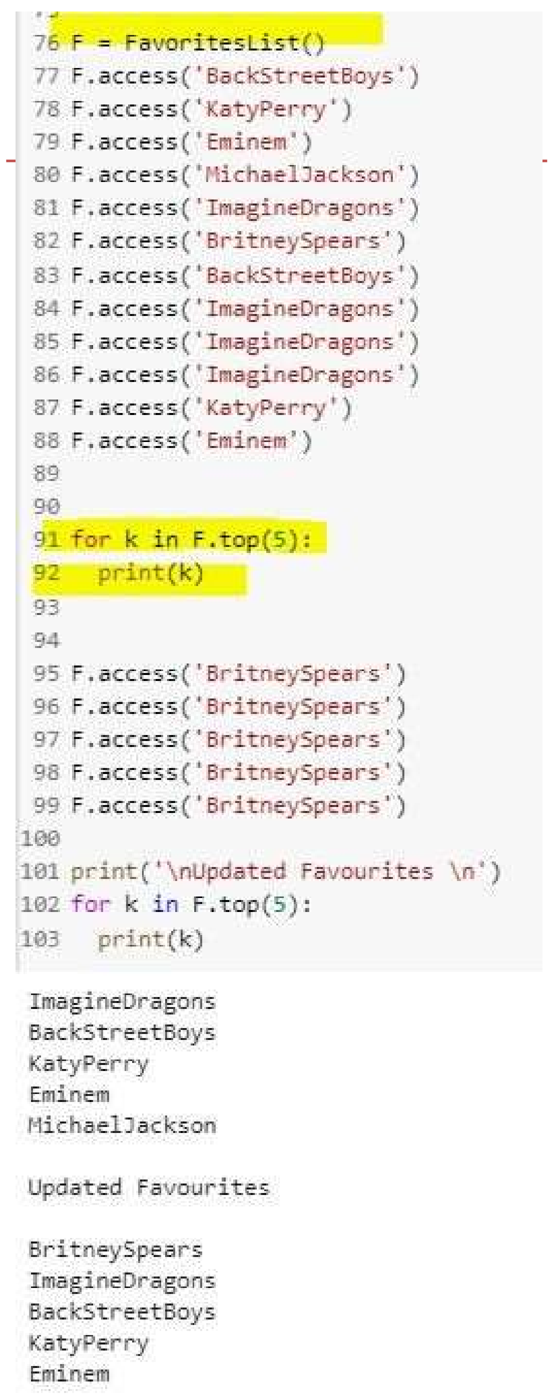
- It uses a **PositionalList** as the underlying data structure.
- Each element of the **PositionalList** is an instance of class **_Item**, having two elements: **_value**, **_count**
- Provides methods to access and remove items from the list and print top favourites
- there are non-public utilities to search for the item and update the list.
- The **top(k)** method returns a sorted list of favourites.
- Every time an item is accessed, its count is incremented and its position within the list is updated by using **_move_up()** method.



```
class FavoritesList:
  '''List of elements ordered from most frequently accessed to least.'''

  #------------------ nested _Item Class -----------------------
  class _Item:
    __slots__ = '_value', '_count'      # streamline memory usage

    def __init__(self, e):
      self._value = e                   # user's element
      self._count = 0                   # access count initially 0

  #---------------- non-public utilities --------------------------
  def _find_position(self, e):
    '''Search for element e and return its Position (or None if not found)'''
    walk = self._data.first()
    while walk is not None and walk.element()._value != e:
      walk = self._data.after(walk)  # move forward
    return walk

  def _move_up(self, p):
    '''Move item at Position p earlier in the list based on access count'''
    if p != self._data.first(): # otherwise, already at top, can't move up
      cnt = p.element()._count
      walk = self._data.before(p) # move up
      if cnt > walk.element()._count:
        while (walk != self._data.first() and
               cnt > self._data.before(walk).element()._count):
          walk = self._data.before(walk) # keep moving upward
        self._data.add_before(walk, self._data.delete(p)) # delete / reinsert
```



```python
  # ------------------ Public Methods ---------------------------------
  def __init__(self):
    ''' Create an empty list of favourites'''
    self._data = PositionalList()       # will be a list of _Item instances

  def __len__(self):
    '''Return the number of entries on the favourite list'''
    return len(self._data)

  def is_empty(self):
    '''Return True if list is empty.'''
    return len(self._data == 0)

  def access(self, e):
    '''Access element e, thereby increasing its access count'''
    p = self._find_position(e)  # try to locate the existing element
    if p is None:
      p = self._data.add_last(self._Item(e))  # if new, place at end
    p.element()._count += 1         # increment its access count
    self._move_up(p)                # consider moving forward

  def remove(self, e):
    ''' Remove element e from the list of favourites'''
    p = self._find_position(e)      # locate existing element
    if p is not None:
      self._data.delete(p)          # delete if found

  def top(self, k):
    '''Generate a sequence of top k elements in terms of access count.'''
    if not 1 <= k <= len(self):
      raise ValueError('Illegal value for k')
    walk = self._data.first()
    for j in range(k):
      item = walk.element()         # element of list is _Item
      yield item._value             # report user's element
      walk = self._data.after(walk) # move forward
```



```python
F = FavoritesList()
F.access('BackStreetBoys')
F.access('KatyPerry')
F.access('Eminem')
F.access('MichaelJackson')
F.access('ImagineDragons')
F.access('BritneySpears')
F.access('BackStreetBoys')
F.access('ImagineDragons')
F.access('ImagineDragons')
F.access('ImagineDragons')
F.access('KatyPerry')
F.access('Eminem')


for k in F.top(5):
  print(k)


F.access('BritneySpears')
F.access('BritneySpears')
F.access('BritneySpears')
F.access('BritneySpears')
F.access('BritneySpears')

print('\nUpdated Favourites \n')
for k in F.top(5):
  print(k)
```

```
ImagineDragons
BackStreetBoys
KatyPerry
Eminem
MichaelJackson

Updated Favourites

BritneySpears
ImagineDragons
BackStreetBoys
KatyPerry
Eminem
```

Using a List with the Move-to-Front Heuristic

- In previous implementation, the method `access(e)` is performed in time proportional to the index of e in the favourite list.
    - That is, if e is the kth most popular element in the favorites list, then accessing it takes O(k) time.
- ***Locality of reference***: once an element is accessed it is more likely to be accessed again in the near future. E.g., web pages visited by an user

- ***Move-to-Front heuristic:*** takes the advantage of locality of reference to improve the performance of access() method.
    - Idea is to move an element all the way to the front of the list each time it is accessed.
    - Our hope, of course, is that this element will be accessed again in the near future.
    - However, the sorting order is lost due to this heuristic.

Consider, for example, a scenario in which we have n elements and the following series of n^2 accesses:

- element 1 is accessed n times
- element 2 is accessed n times
- $\cdots$
- element n is accessed n times.

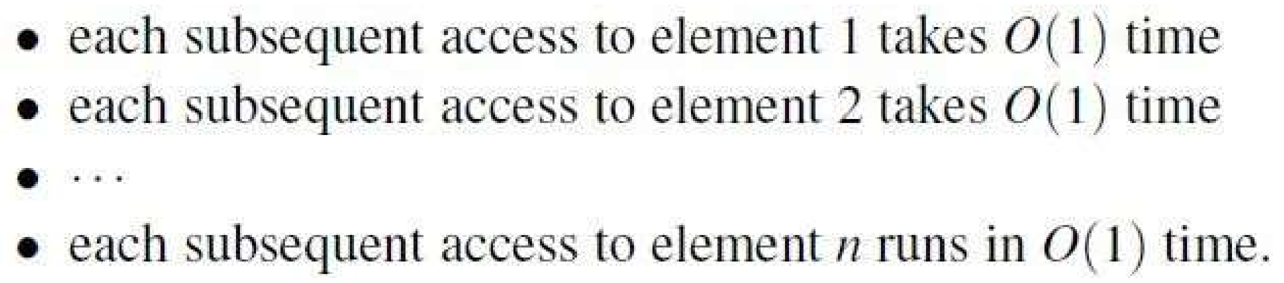
If we store the elements sorted by their access counts, inserting each element the first time it is accessed, then

- each access to element 1 runs in $O(1)$ time
- each access to element 2 runs in $O(2)$ time
- $\cdots$
- each access to element n runs in $O(n)$ time.

Thus, the total time for performing the series of accesses is proportional to

$$n + 2n + 3n + \cdots + n \cdot n = n(1 + 2 + 3 + \cdots + n) = n \cdot \frac{n(n+1)}{2}, \qquad \sim O(n^3)$$

If we use move-to-front heuristic, inserting each element the first time it is accessed, then



- each subsequent access to element 1 takes $O(1)$ time
- each subsequent access to element 2 takes $O(1)$ time
- $\cdots$
- each subsequent access to element n runs in $O(1)$ time.

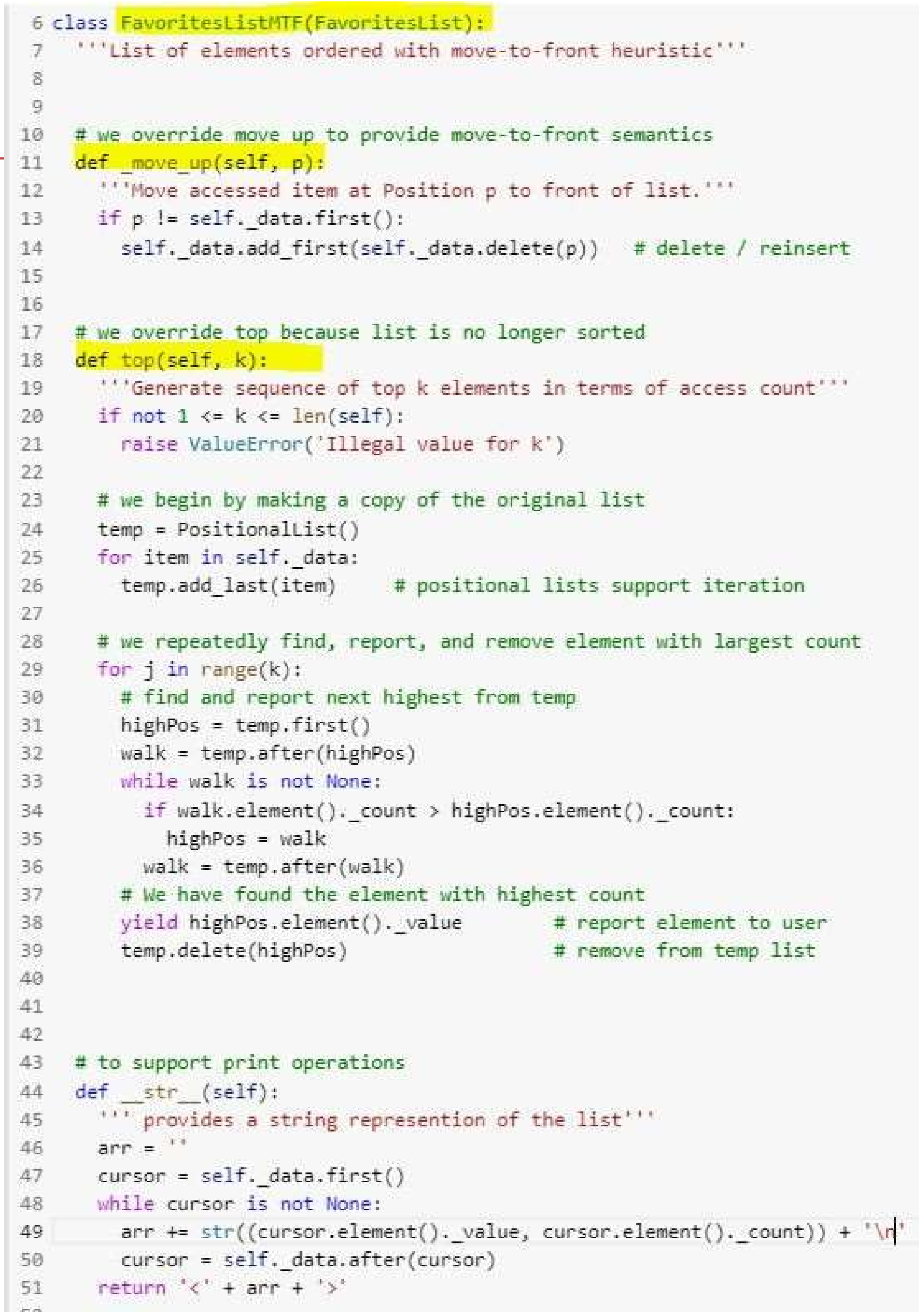
- So the running time for performing all the accesses in this case is O(n^2).
- So, in this case, move-to-front heuristic is faster.
- This effect is not universal. This heuristic might be slower in other sequences.

The trade-offs with the Move-to-Front Heuristic

- MTF version does not maintain the elements of the favourite list ordered by their access counts.
- When we are asked to find the k most accessed elements, we need to search for them.
- We will implement the `top(k)` method as follows:
    - We copy all entries of our favorites list into another list, named temp.
    - We scan the temp list k times. In each scan, we find the entry with the largest access count, remove this entry from temp, and report it in the results.

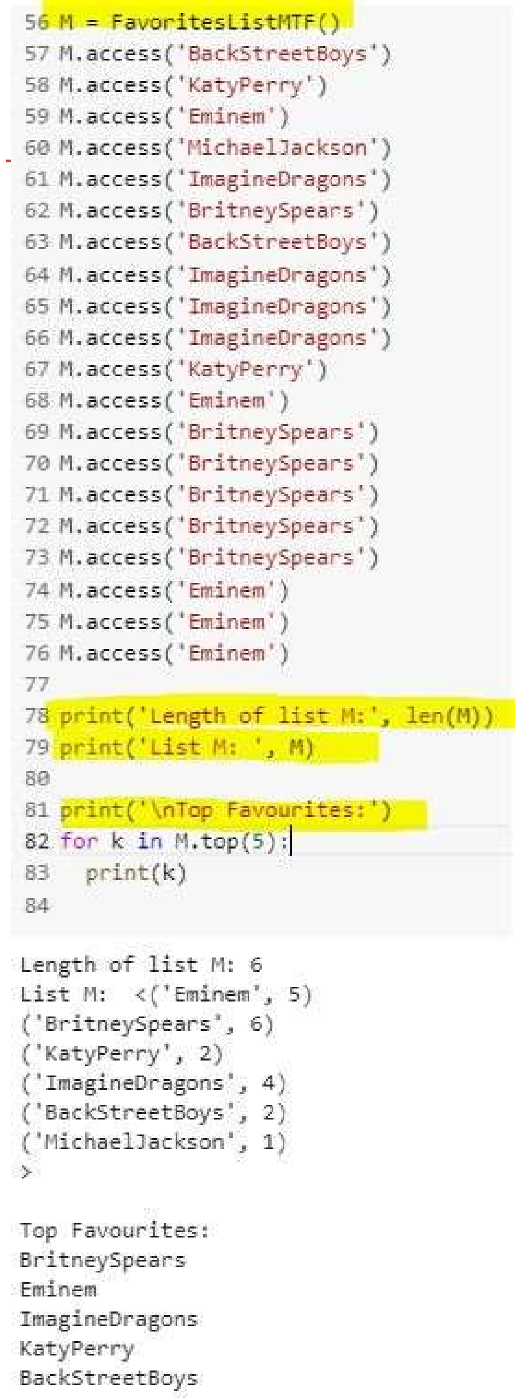
- This implementation of method top takes O(kn) time.
    - Thus, when k is a constant, method top runs in O(n) time. However, if k is proportional to n, then top runs in O(n^2) time.
    - Example: If we want a top 25% of the list.

- It possible to implement top(k) method in $O(n + k \log n)$ time with ***Priority Queues***.
- We could easily achieve $O(n \log n)$ time if we use a standard sorting algorithm to reorder the temporary list before reporting the top k items.



```python
class FavoritesListMTF(FavoritesList):
  '''List of elements ordered with move-to-front heuristic'''


  # we override move up to provide move-to-front semantics
  def _move_up(self, p):
    '''Move accessed item at Position p to front of list.'''
    if p != self._data.first():
      self._data.add_first(self._data.delete(p))   # delete / reinsert


  # we override top because list is no longer sorted
  def top(self, k):
    '''Generate sequence of top k elements in terms of access count'''
    if not 1 <= k <= len(self):
      raise ValueError('Illegal value for k')

    # we begin by making a copy of the original list
    temp = PositionalList()
    for item in self._data:
      temp.add_last(item)      # positional lists support iteration

    # we repeatedly find, report, and remove element with largest count
    for j in range(k):
      # find and report next highest from temp
      highPos = temp.first()
      walk = temp.after(highPos)
      while walk is not None:
        if walk.element()._count > highPos.element()._count:
          highPos = walk
        walk = temp.after(walk)
      # We have found the element with highest count
      yield highPos.element()._value        # report element to user
      temp.delete(highPos)                  # remove from temp list



  # to support print operations
  def __str__(self):
    ''' provides a string represention of the list'''
    arr = ''
    cursor = self._data.first()
    while cursor is not None:
      arr += str((cursor.element()._value, cursor.element()._count)) + '\n'
      cursor = self._data.after(cursor)
    return '<' + arr + '>'
```

- We modify _move_up() method to implement move-to-front heuristic
- The top() method is also modified for sorting the list to be displayed



```python
M = FavoritesListMTF()
M.access('BackStreetBoys')
M.access('KatyPerry')
M.access('Eminem')
M.access('MichaelJackson')
M.access('ImagineDragons')
M.access('BritneySpears')
M.access('BackStreetBoys')
M.access('ImagineDragons')
M.access('ImagineDragons')
M.access('ImagineDragons')
M.access('KatyPerry')
M.access('Eminem')
M.access('BritneySpears')
M.access('BritneySpears')
M.access('BritneySpears')
M.access('BritneySpears')
M.access('BritneySpears')
M.access('Eminem')
M.access('Eminem')
M.access('Eminem')

print('Length of list M:', len(M))
print('List M: ', M)

print('\nTop Favourites:')
for k in M.top(5):
  print(k)
```

```
Length of list M: 6
List M:  <('Eminem', 5)
('BritneySpears', 6)
('KatyPerry', 2)
('ImagineDragons', 4)
('BackStreetBoys', 2)
('MichaelJackson', 1)
>

Top Favourites:
BritneySpears
Eminem
ImagineDragons
KatyPerry
BackStreetBoys
```

Link-Based Vs Array-Based Sequences

Advantages of Array-based Sequences:

- Arrays provide O(1)-time access to an element based on an integer index. In contrast, Linked-list provides O(k) time to access kth element or O(n-k) time if traversing backward in a doubly linked list.

- Operations with equivalent asymptotic bounds typically run a constant factor more efficiently with an array-based structure versus a linked structure.

- Array-based representations typically use proportionally less memory than linked structures. Array-based containers need space only for elements. With linked lists, memory is required for storing not only elements but also references to nodes.

Advantages of Link-Based Sequences:

- Link-based structures provide worst-case time bounds for their operations. This is in contrast to amortized bounds associated with the expansion or contraction of dynamic arrays - essential for real-time systems.

- Link-based structures support O(1)-time insertions and deletions at arbitrary positions. This is in stark contrast to array-based sequences. For instance, insert() or pop() methods of python list class takes O(n-k+10) for operating at index k as it requires a loop to shift all subsequent elements.

Summary

- We cover the following topics in this lecture:
  - Limitations of Array-based Sequences - Why Linked-lists are required?
  - Different types of Linked lists: Singly linked-lists, circularly linked-lists, doubly linked-lists.
  - We provide an implemented of double ended queues (deques) using linked list.
  - We study an implementation of a Linked List called PostionalLists where node references could be used a location indicator within the list.
  - We implement one case study of maintaining Favourite List based on its access count.
  - We study the relative comparison between array-based and link-based sequences