07

Linked Lists

Chapter 7

Linked Lists

The problems of list class (recall it uses dynamic array)

- The length of internal array might be lager than needed
- Amortize cost for append might unacceptable for certain cases
- Insertions and deletions for internal posistions is expensive

Linked Lists

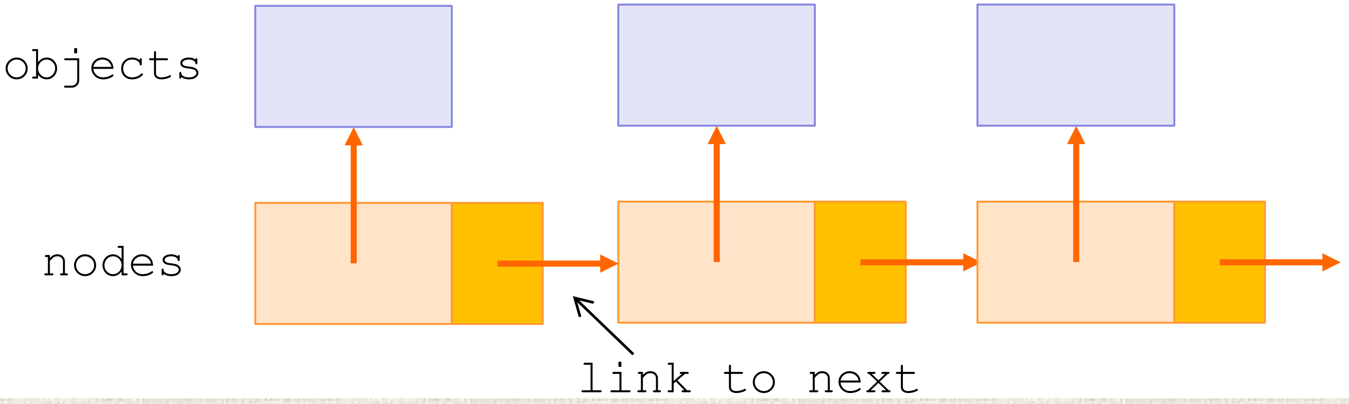
Linked List ADT overcomes all these problems

Dynamic array comprises consecutive bytes from memory

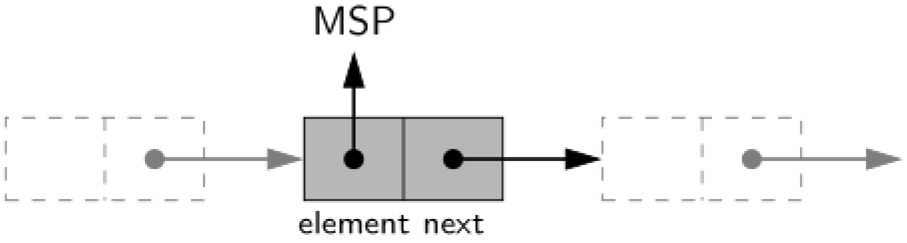
Linked list is comprised of "nodes" which can be anywhere in the memory

"Nodes" are connected to each other with one or more links

Each node holds pointers to data and other nodes



Singly Linked List



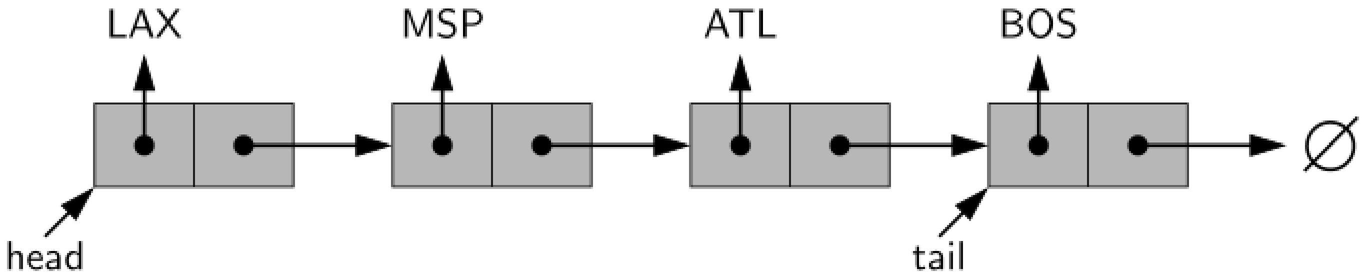
Collection of nodes each referencing a single node

Represents a linear sequence

References can point to

- Next node
- Any arbitrary object (a dict, a list, a class instance, an int, etc.)

Singly Linked List



First and last nodes are called **head** and **tail**.

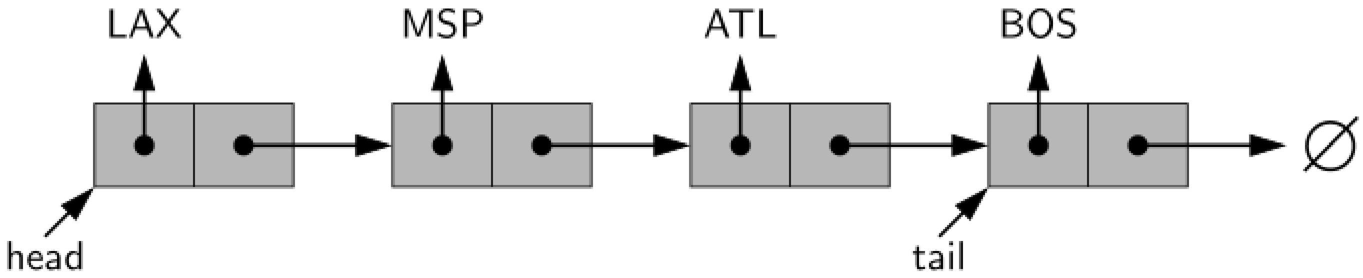
Each link between two nodes is called **next**.

next is actually just a reference.

tail node's next reference points to **None**.

Visiting each node of a linked list starting from head until reaching tail is called traversing (also known as link/pointer hopping).

Singly Linked List

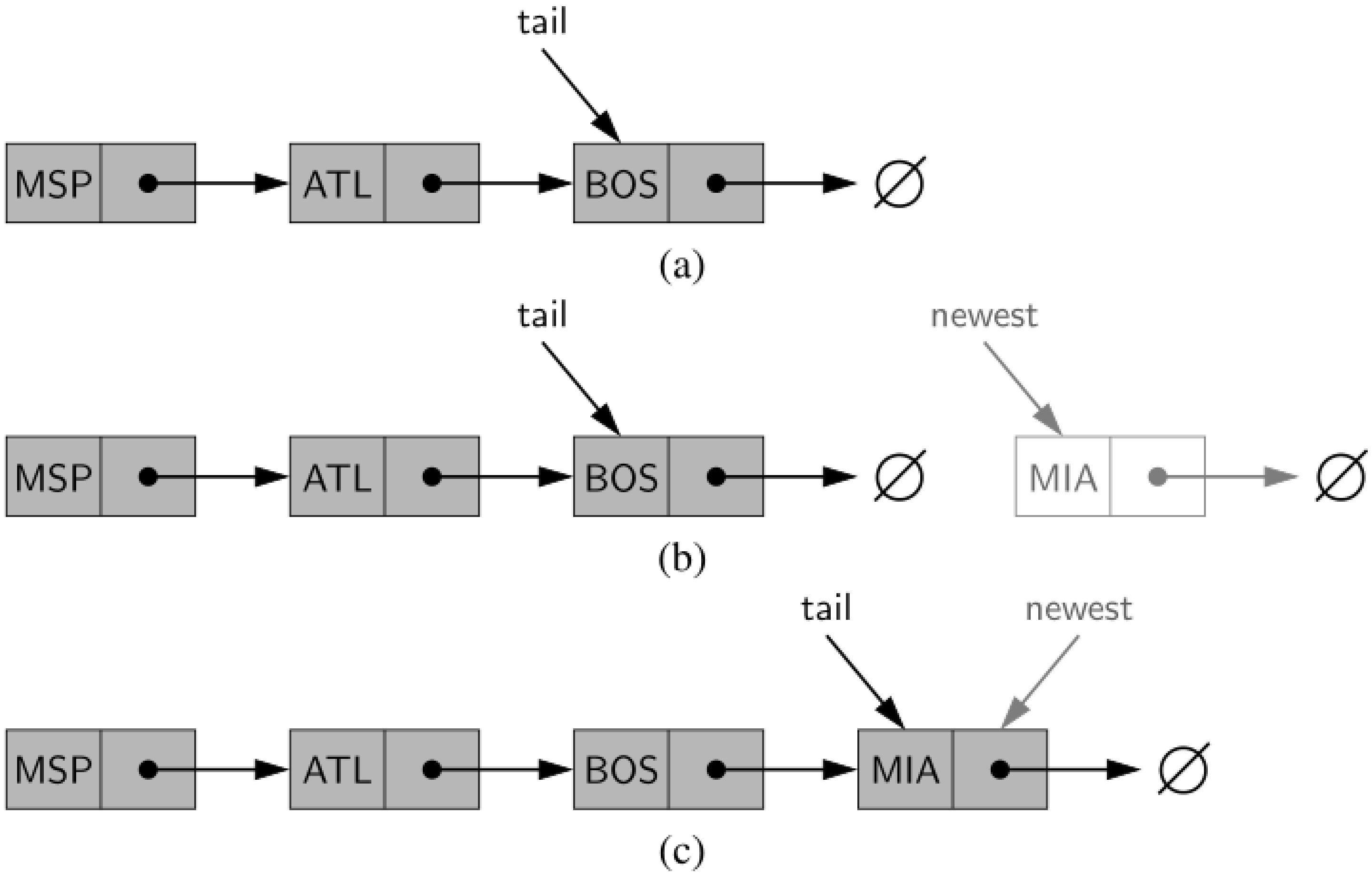


There are two class types:

Node (representing node, has at least two refs: To data object and to next node.

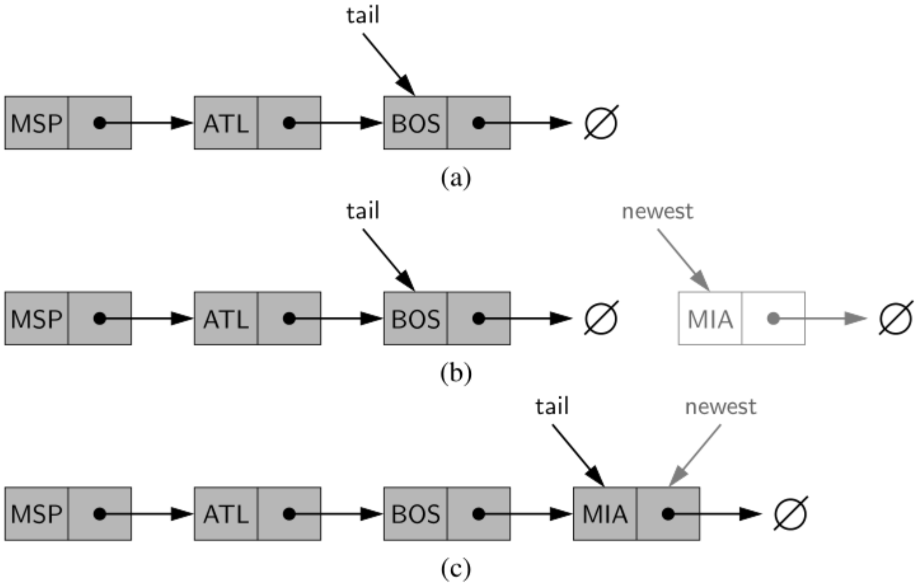
LinkedList (manages the linked list operations, has to have a reference to the head of the linked list, optionally there might a tail ref and a counter for storing the number of items in the list.

Singly Linked List - Insertion

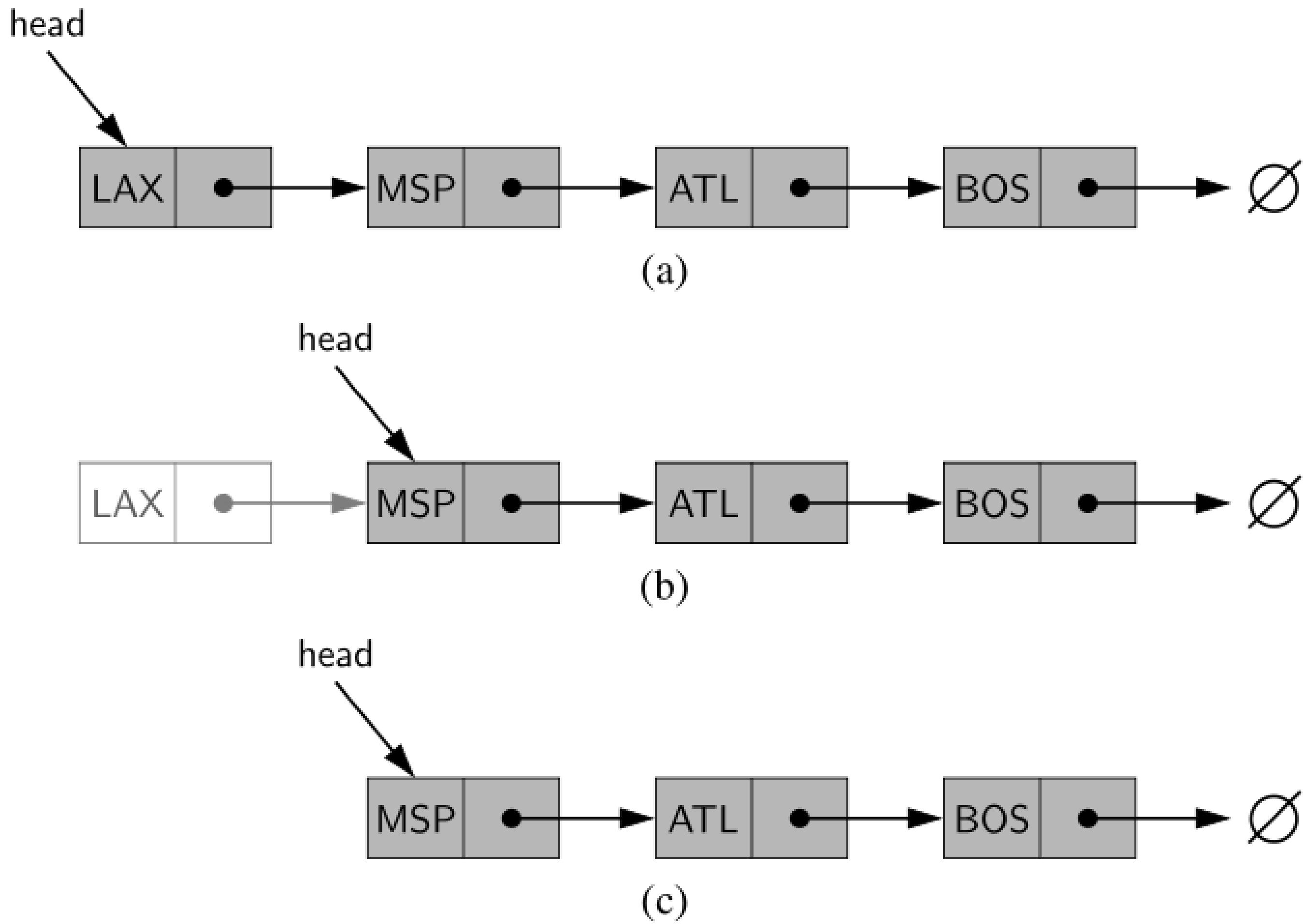


Singly Linked List - Insertion

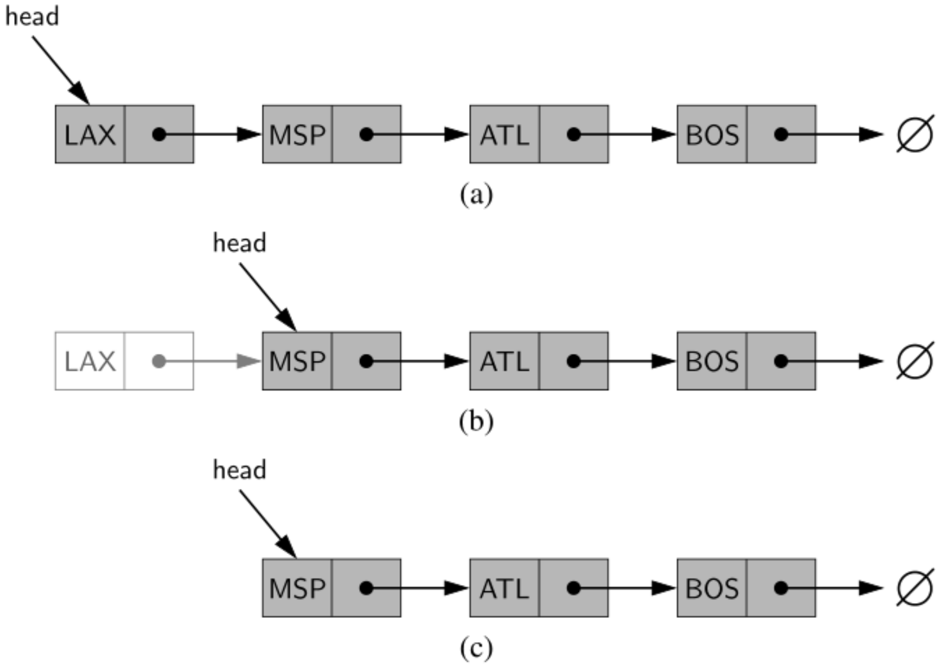
```
Algorithm add_last(L, e):
    newest = Node(e)
    newest.next = None
    L.tail.next = newest
    L.tail = newest
    L.size = L.size + 1
```



Singly Linked List - Removing First



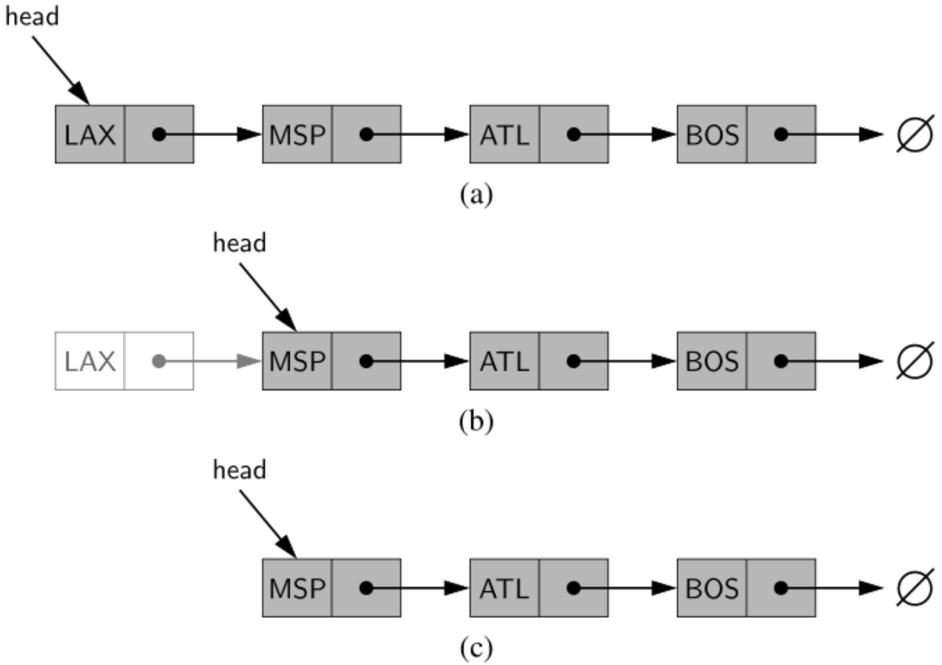
Singly Linked List - Removing First



```
Algorithm remove_first(L):
    if L.head is None then
        Indicate an error: the list is empty.
    L.head = L.head.next
    L.size = L.size − 1
```

Singly Linked List - Removing First

What about deleting the last node?

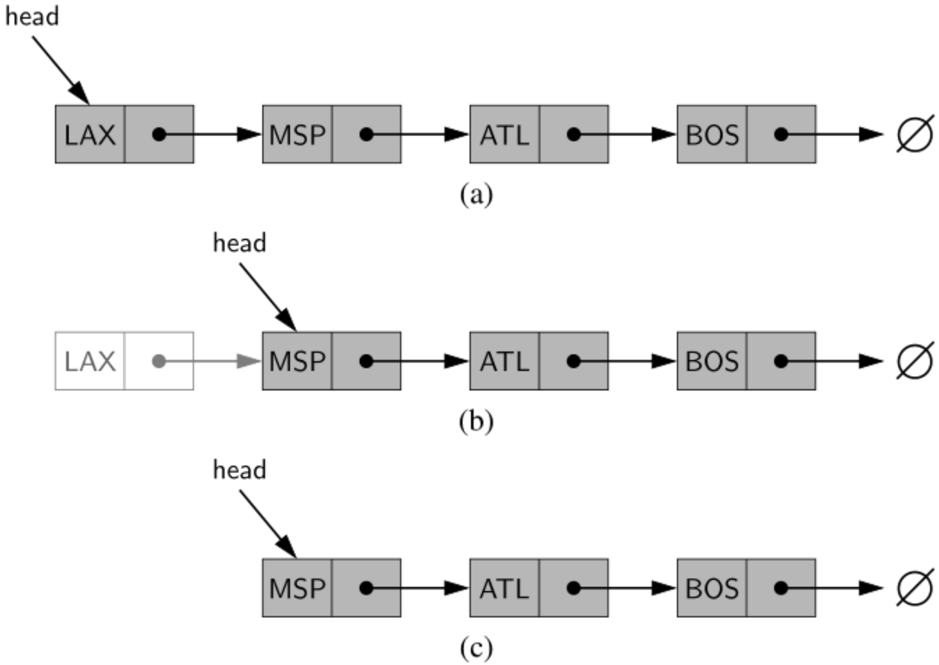


Singly Linked List - Removing First

What about deleting the last node?

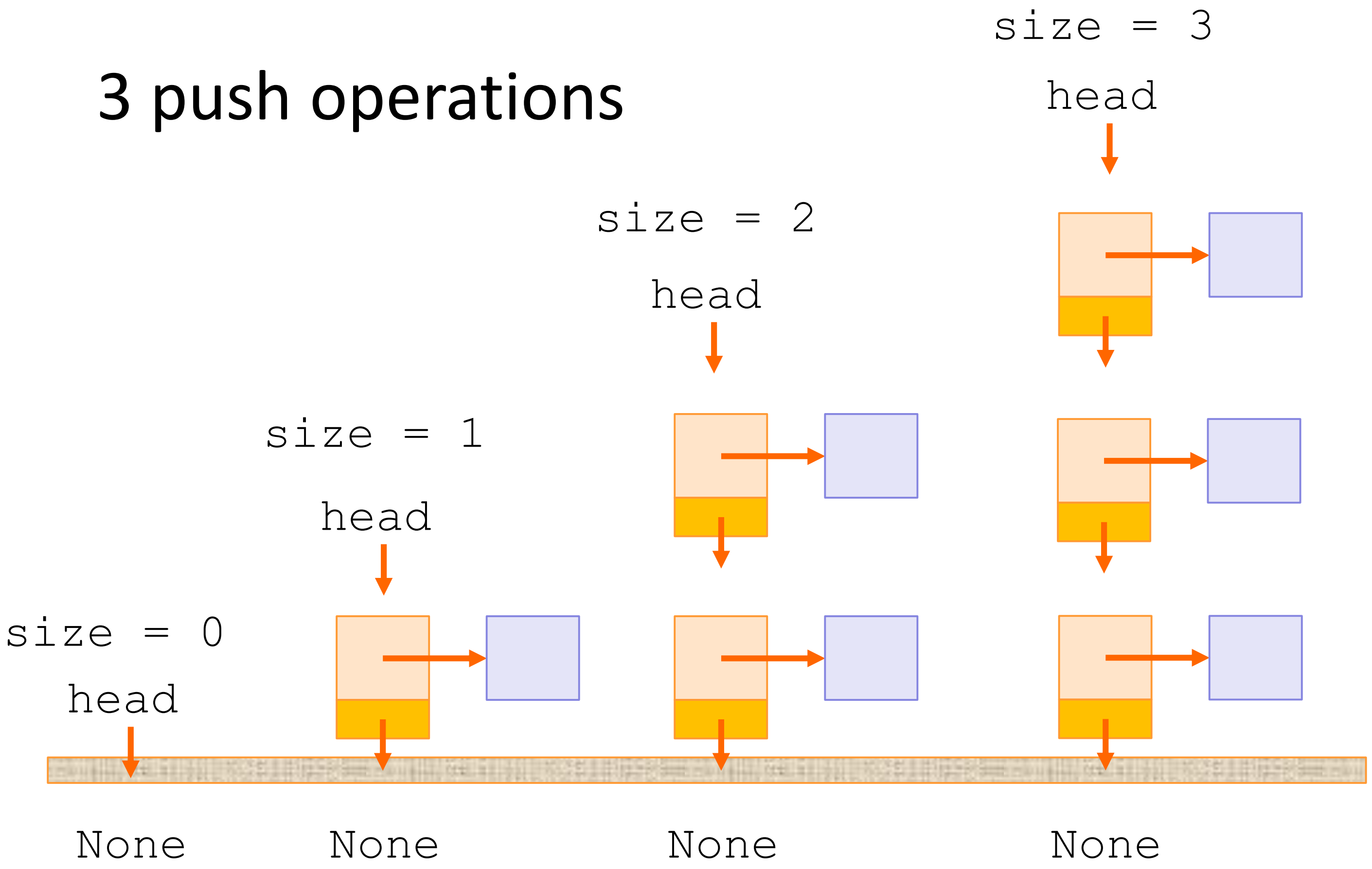
That would be cost operation: Traverse all the list starting from head...

Efficient Solution:
Doubly linked list



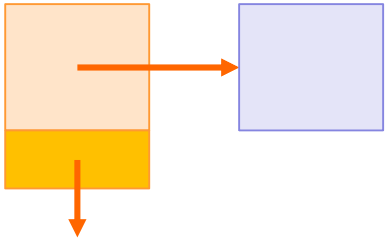
Implementing a Stack with Singly Linked List

3 push operations



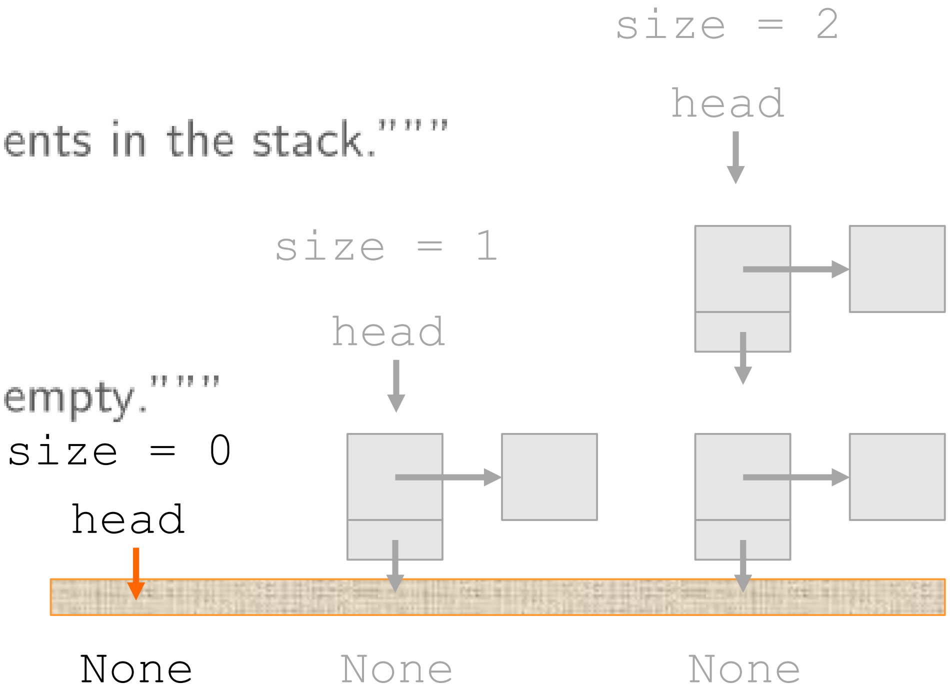
Implementing a Stack with Singly Linked List

```
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 fields
      self._element = element              # reference to user's element
      self._next = next                    # reference to next node
```



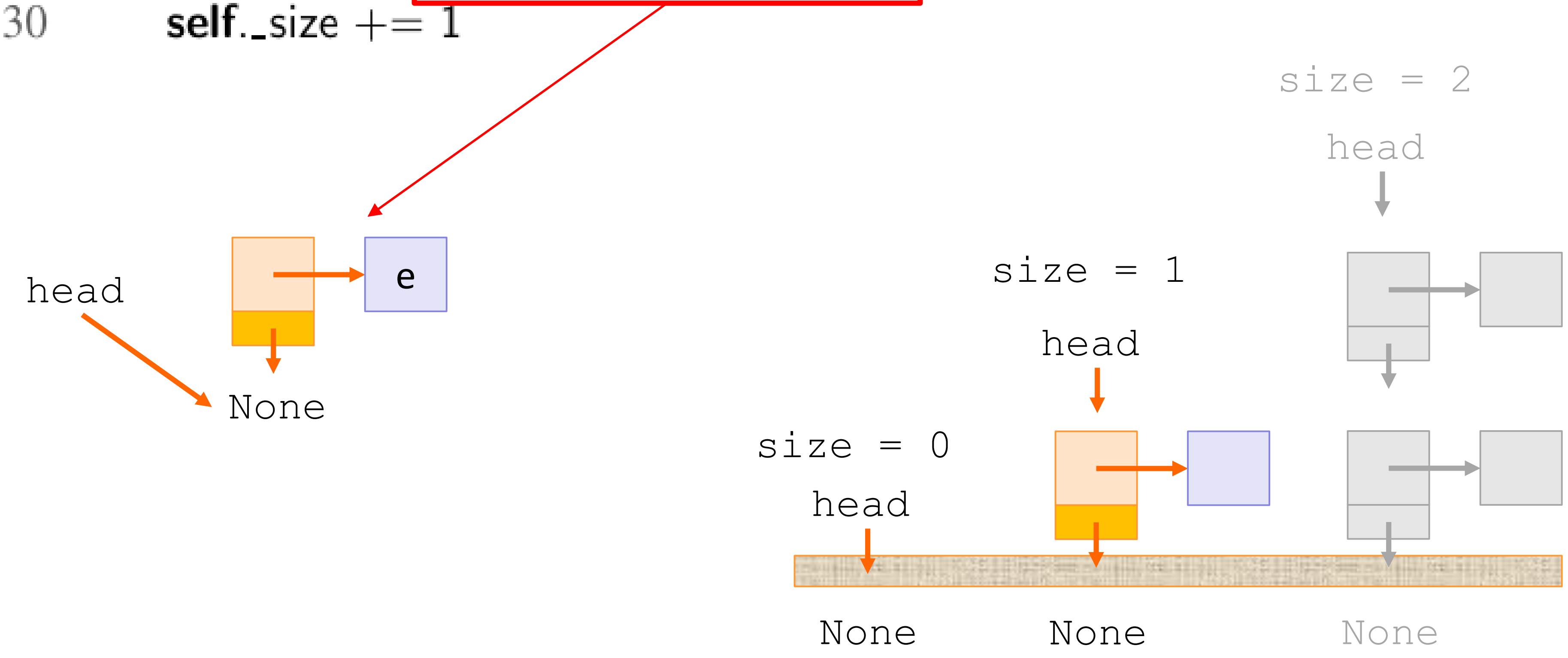
Implementing a Stack with Singly Linked List

```
#------------------------------ stack methods ------------------------------
def __init__(self):
  """Create an empty stack."""
  self._head = None                          # reference to the head node
  self._size = 0                             # number of stack elements

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

def is_empty(self):
  """Return True if the stack is empty."""
  return self._size == 0
```



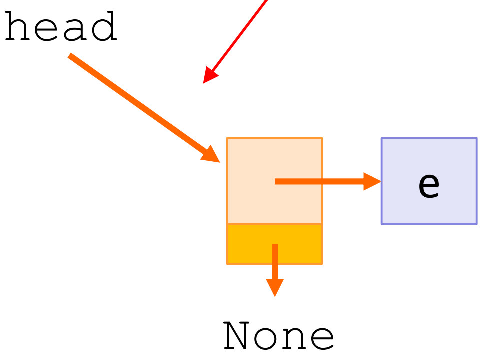
Implementing a Stack with Singly Linked List

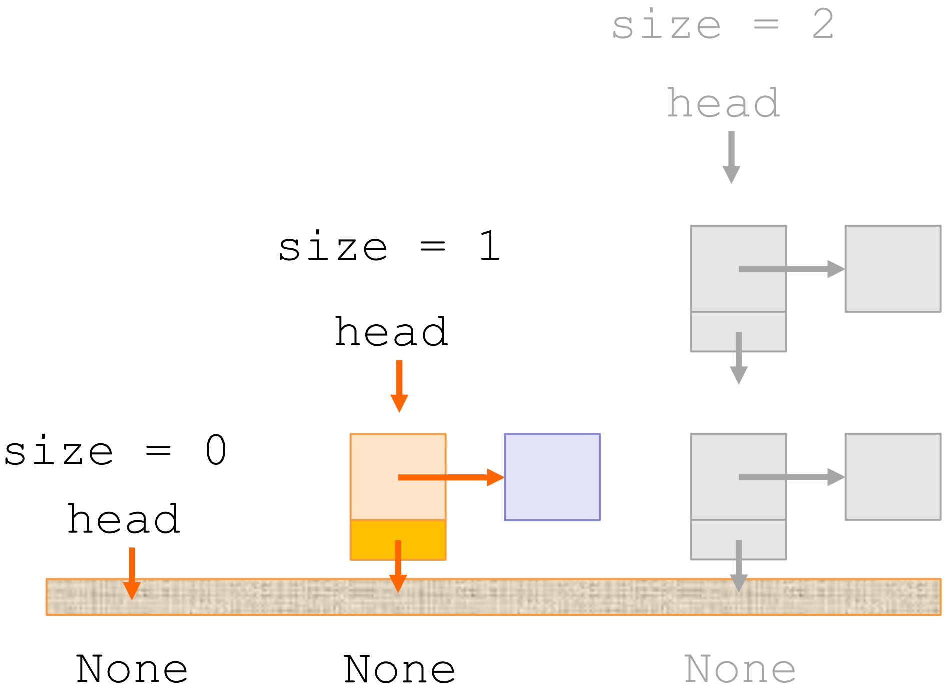
```
def push(self, e):
  """Add element e to the top of the stack."""
  self._head = self._Node(e, self._head)     # create and link a new node
  self._size += 1
```



Implementing a Stack with Singly Linked List

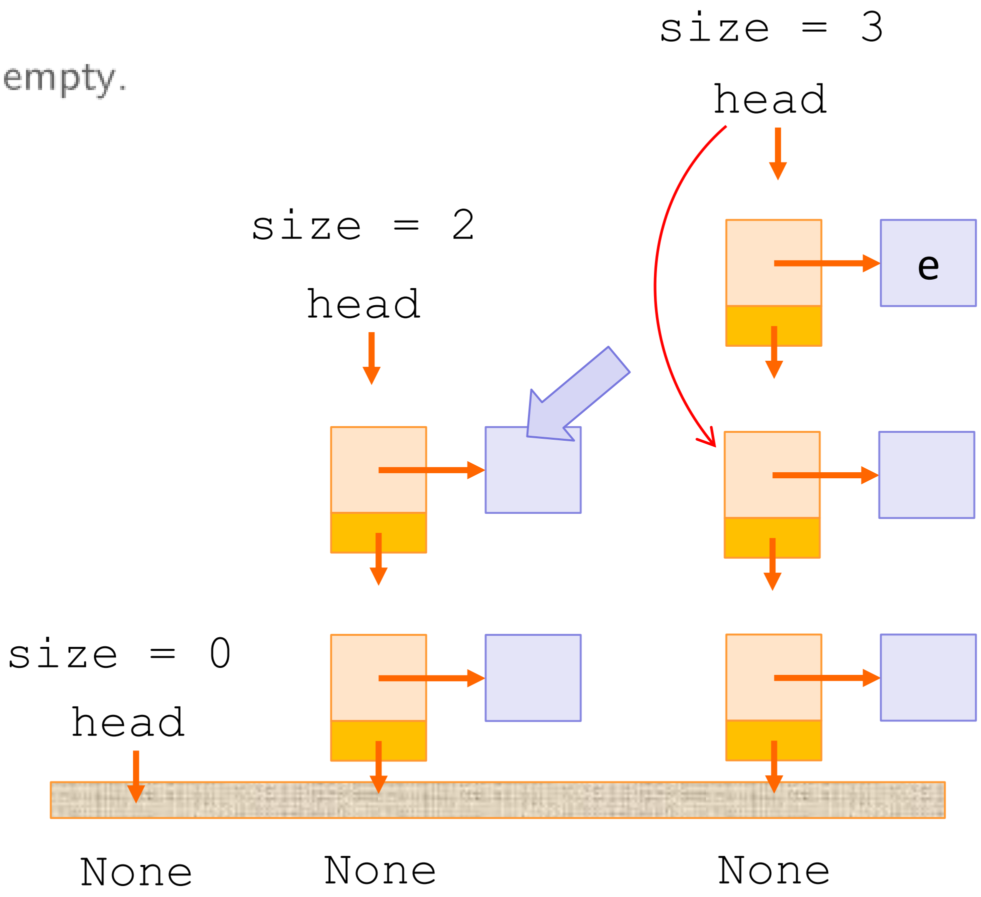
```
def push(self, e):
    """Add element e to the top of the stack."""
    self._head = self._Node(e, self._head)     # create and link a new node
    self._size += 1
```





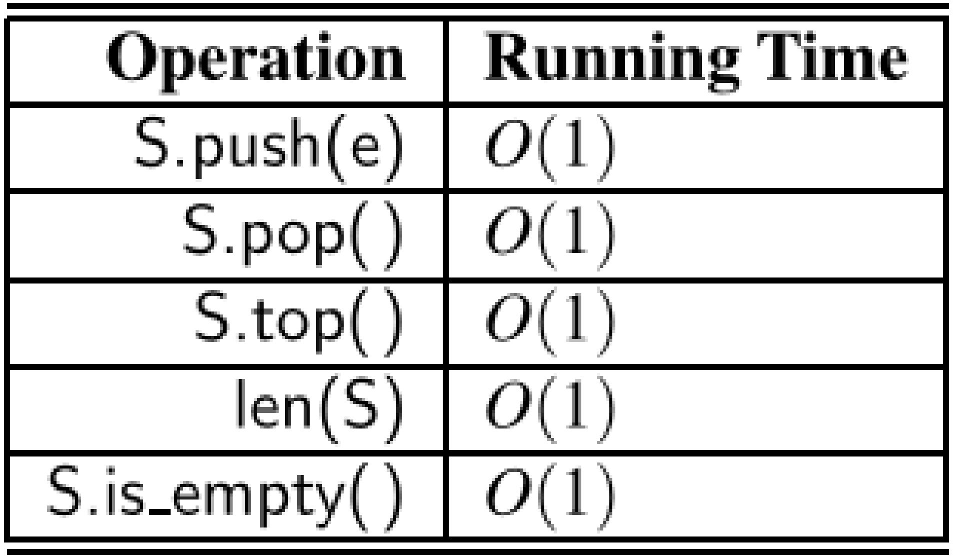
Implementing a Stack with Singly Linked List

```
def pop(self):
  """Remove and return the element from the top of the stack (i.e., LIFO).

  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
  self._size −= 1
  return answer
```



Implementing a Stack with Singly Linked List

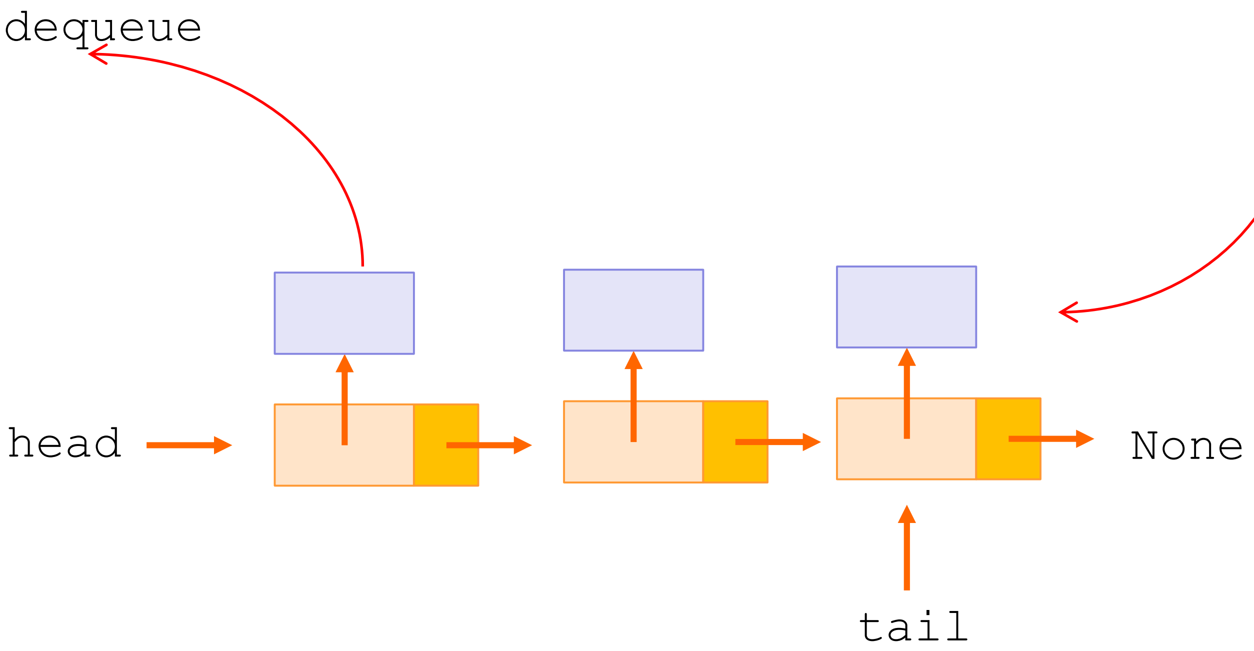
Complexity of LinkedStack Operations



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

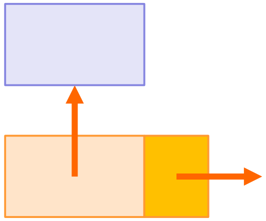
Implementing a Queue with Singly Linked List

3 enqueue operations



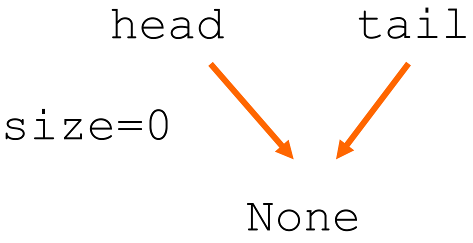
Implementing a Queue with Singly Linked List

```
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 fields
      self._element = element               # reference to user's element
      self._next = next                     # reference to next node
```



Implementing a Queue with Singly Linked List

```
  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
```



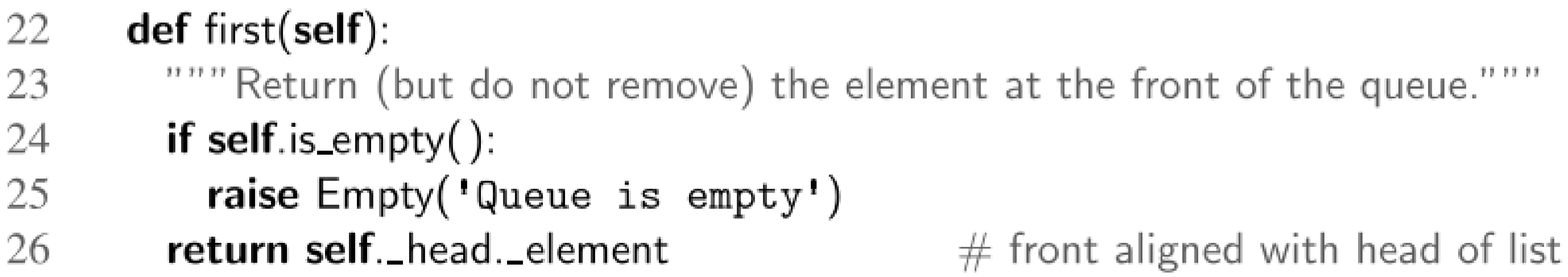
head

tail

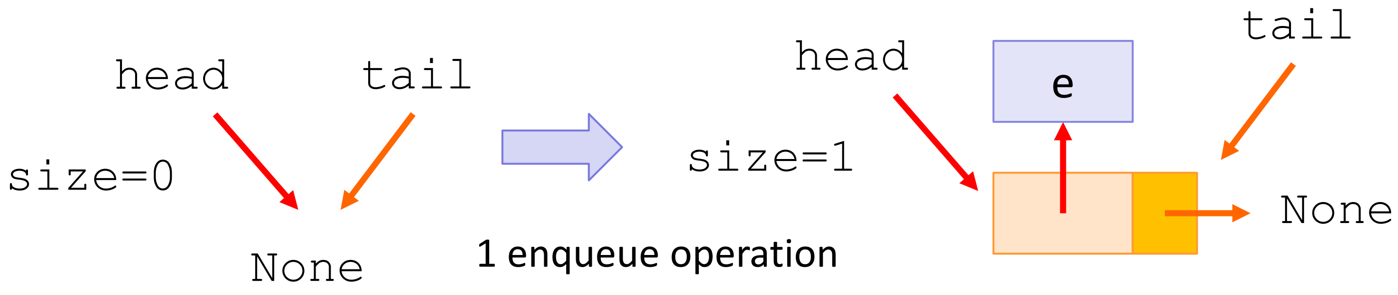
size=0

None

Implementing a Queue with Singly Linked List



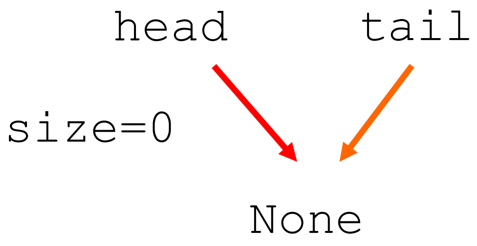
```
  def first(self):
    """Return (but do not remove) the element at the front of the queue."""
    if self.is_empty():
      raise Empty('Queue is empty')
    return self._head._element                  # front aligned with head of list
```



Implementing a Queue with Singly Linked List

```
def dequeue(self):
  """Remove and return the first element of the queue (i.e., 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
  self._size −= 1
  if self.is_empty():
    self._tail = None
  return answer
```

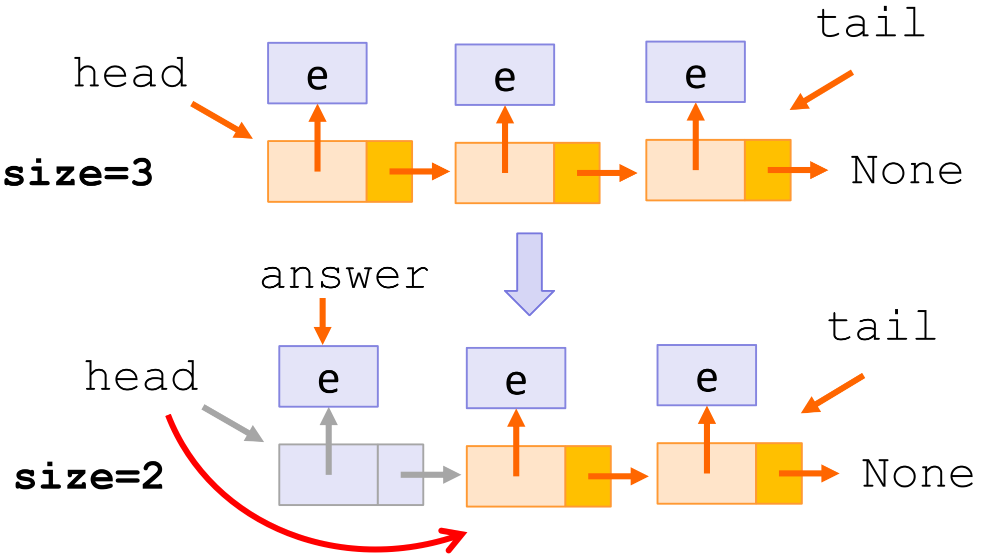
Case 1: Queue is empty



Implementing a Queue with Singly Linked List

```
def dequeue(self):
  """Remove and return the first element of the queue (i.e., 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
  self._size -= 1
  if self.is_empty():
    self._tail = None
  return answer
```

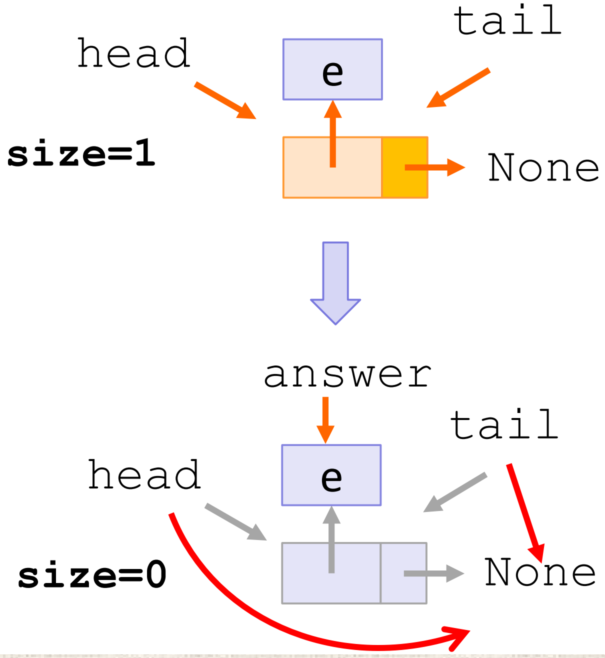
Case 2: After dequeue there are still some items



Implementing a Queue with Singly Linked List

```
def dequeue(self):
  """Remove and return the first element of the queue (i.e., 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
  self._size −= 1
  if self.is_empty():
    self._tail = None
  return answer
```

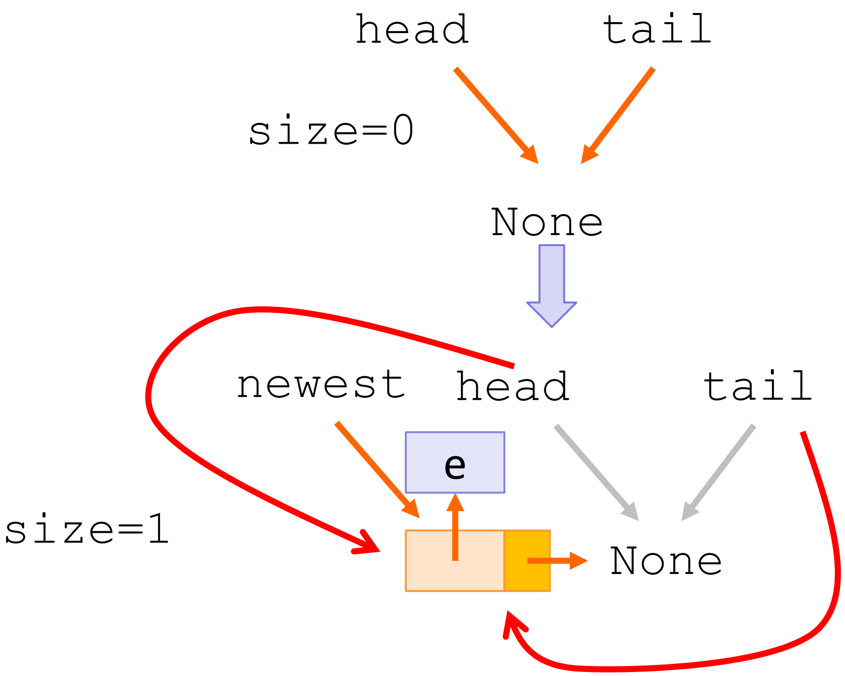
Case 3: After dequeue queue becomes empty



Implementing a Queue with Singly Linked List

```python
    def enqueue(self, e):
        """Add an element to the back of queue."""
        newest = self._Node(e, None)
        if self.is_empty():
            self._head = newest
        else:
            self._tail._next = newest
        self._tail = newest
        self._size += 1
```

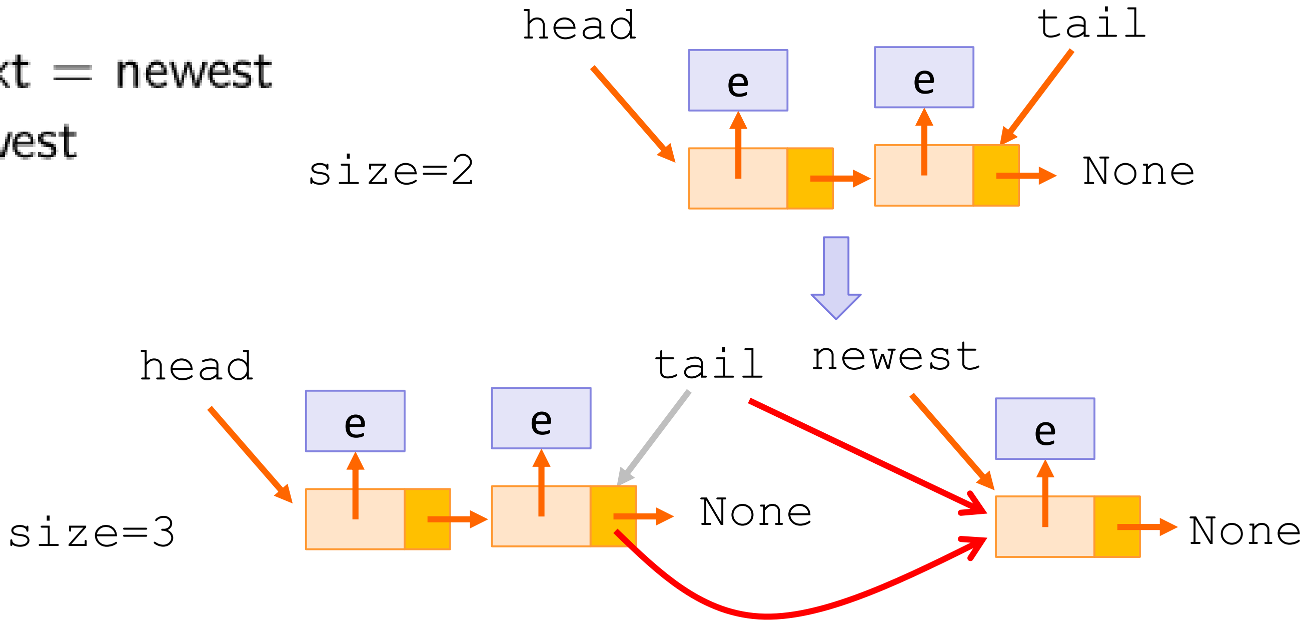
Case 1: Queue is empty



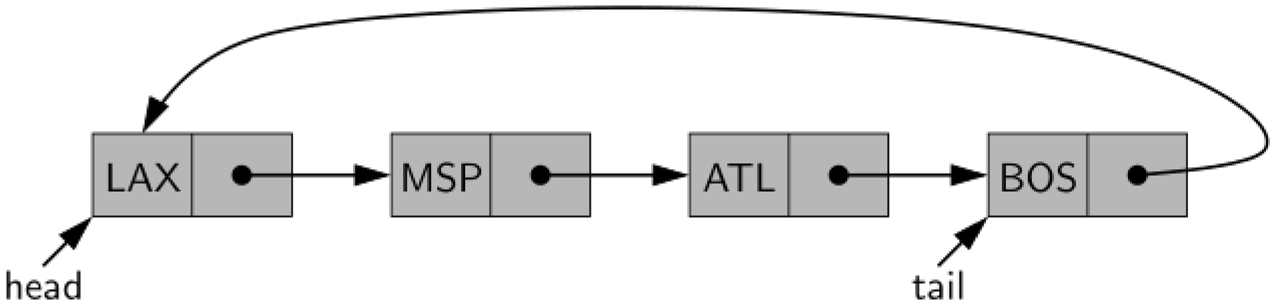
Implementing a Queue with Singly Linked List

```
def enqueue(self, e):
    """Add an element to the back of queue."""
    newest = self._Node(e, None)
    if self.is_empty():
        self._head = newest
    else:
        self._tail._next = newest
    self._tail = newest
    self._size += 1
```

Case 1: Queue is not empty

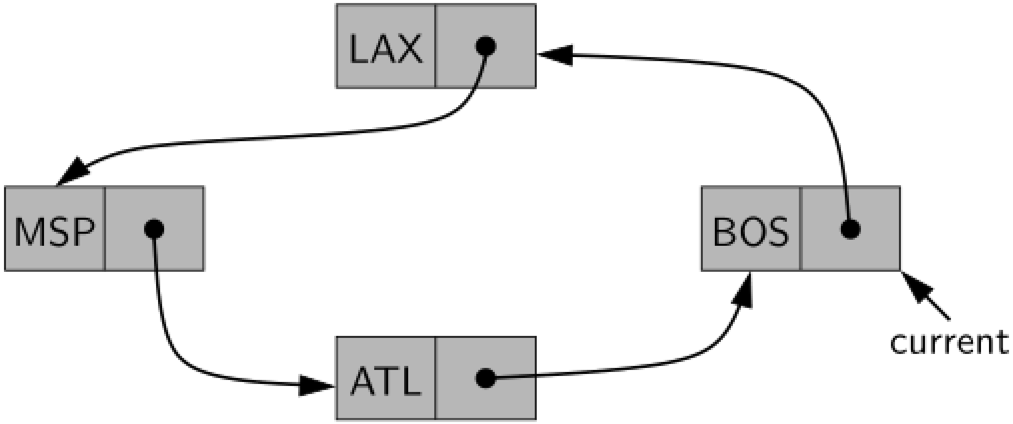


Circularly Linked List



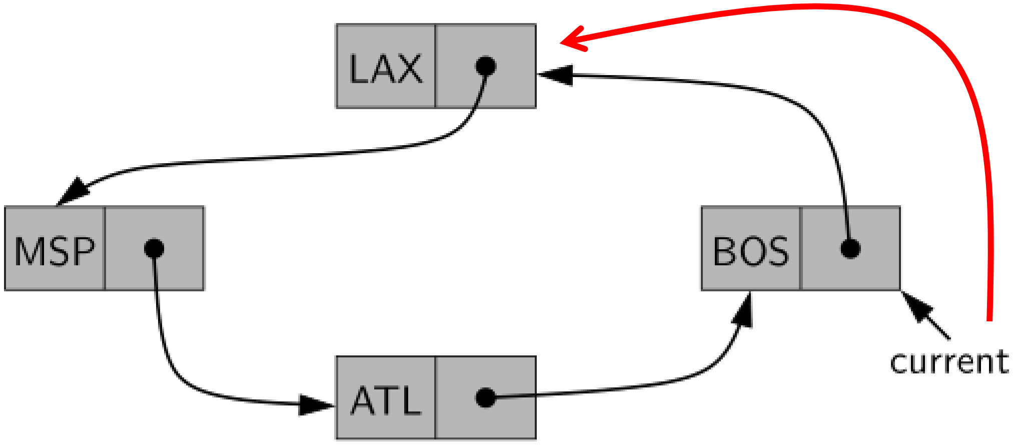
A more suitable data structure for "circular data," where beginning or end is not a concern.

There might be cases where all we are interested is the "current" item.



Circularly Linked List

There might be cases where all we are interested is the "current" item.

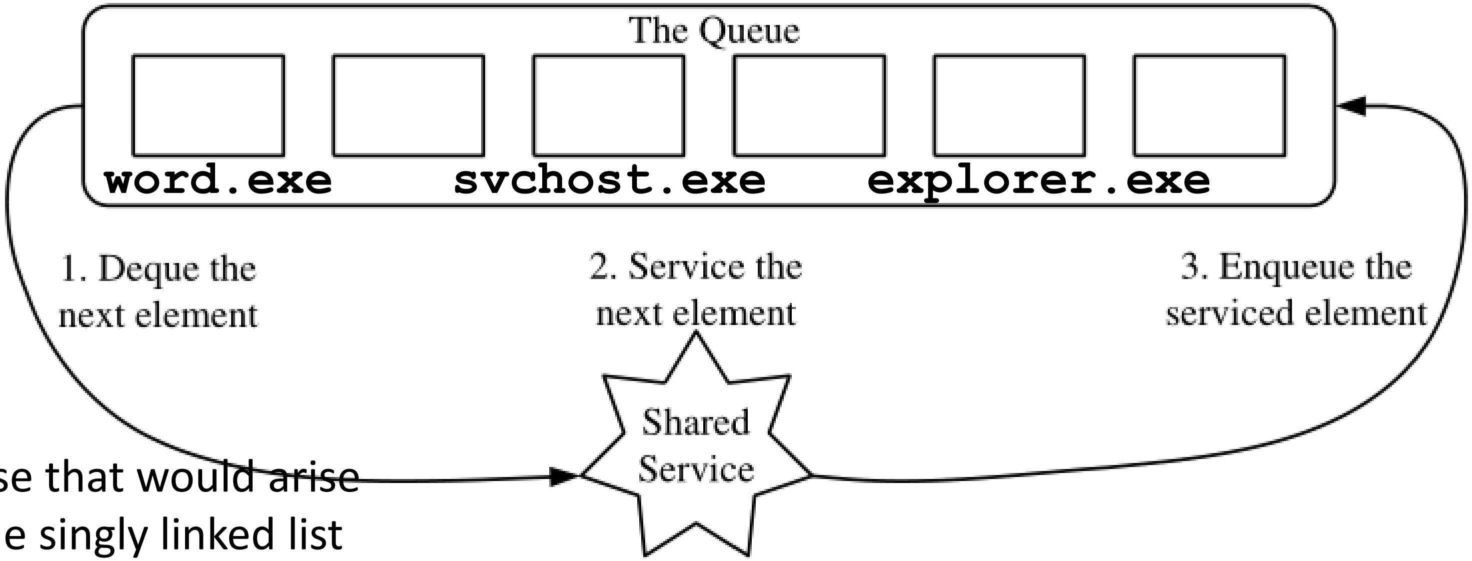


We can easily traverse through the list with:

```
current = current.next
```

Circularly Linked List

A usecase: Round-Robin Scheduler


The Queue
word.exe
svchost.exe
explorer.exe
1. Deque the next element
2. Service the next element
3. Enqueue the serviced element
Shared Service

CPU

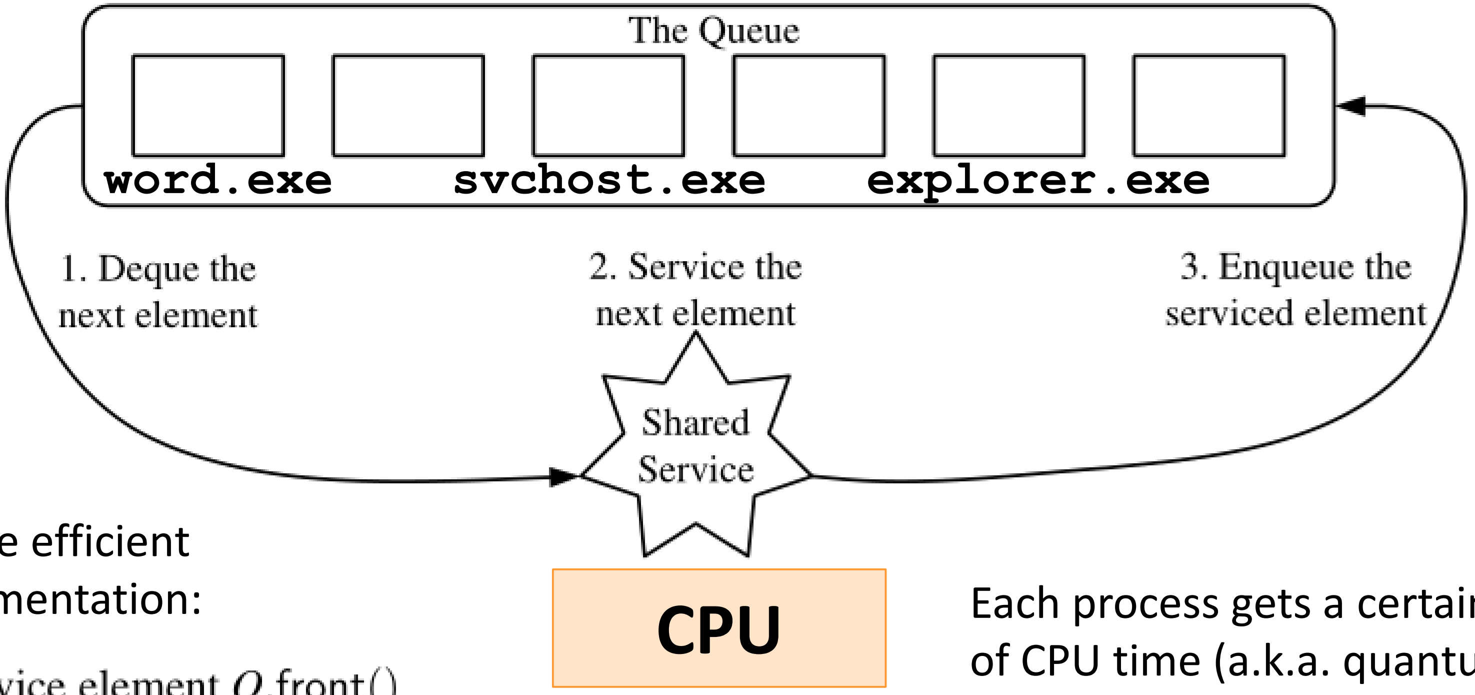
The case that would arise with the singly linked list implementation:

1. $e = Q$.dequeue()
2. Service element e
3. Q.enqueue(e)

Each process gets a certain amount of CPU time (a.k.a. quantum) (e.g., a couple nanoseconds) in turn.

Circularly Linked List

A usecase: Round-Robin Scheduler

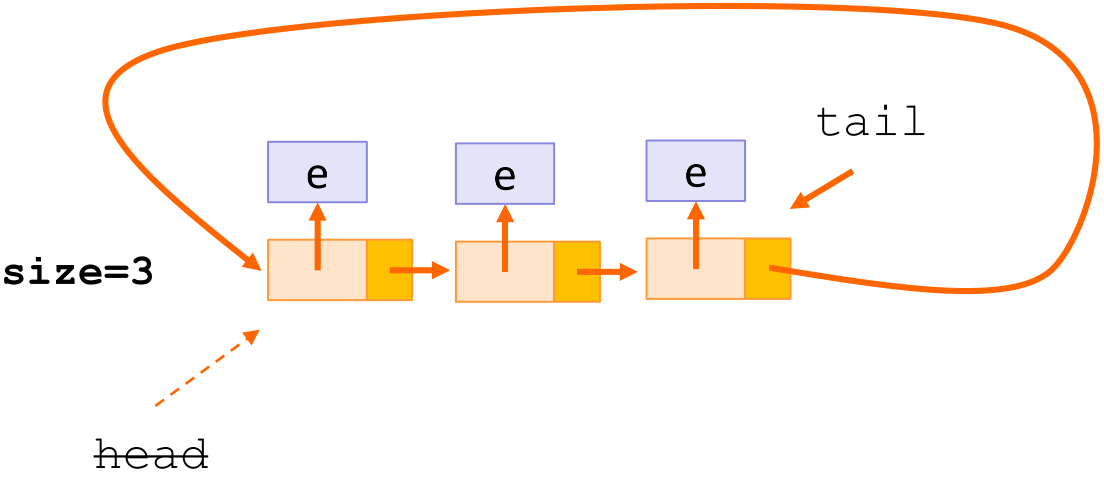

The Queue
word.exe
svchost.exe
explorer.exe
1. Deque the next element
2. Service the next element
3. Enqueue the serviced element
Shared Service
CPU

A more efficient implementation:

1. Service element Q.front()
2. Q.rotate() i.e., switch to next element

Each process gets a certain amount of CPU time (a.k.a. quantum) (e.g., a couple nanoseconds) in turn.

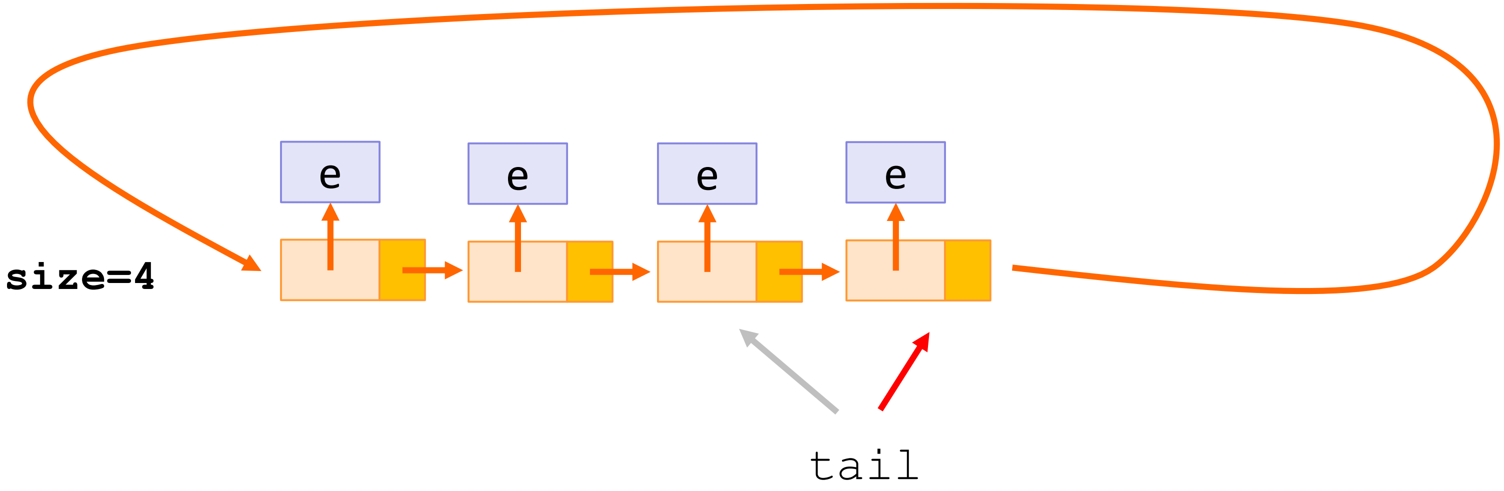
Circularly Linked List



No need for head pointer.

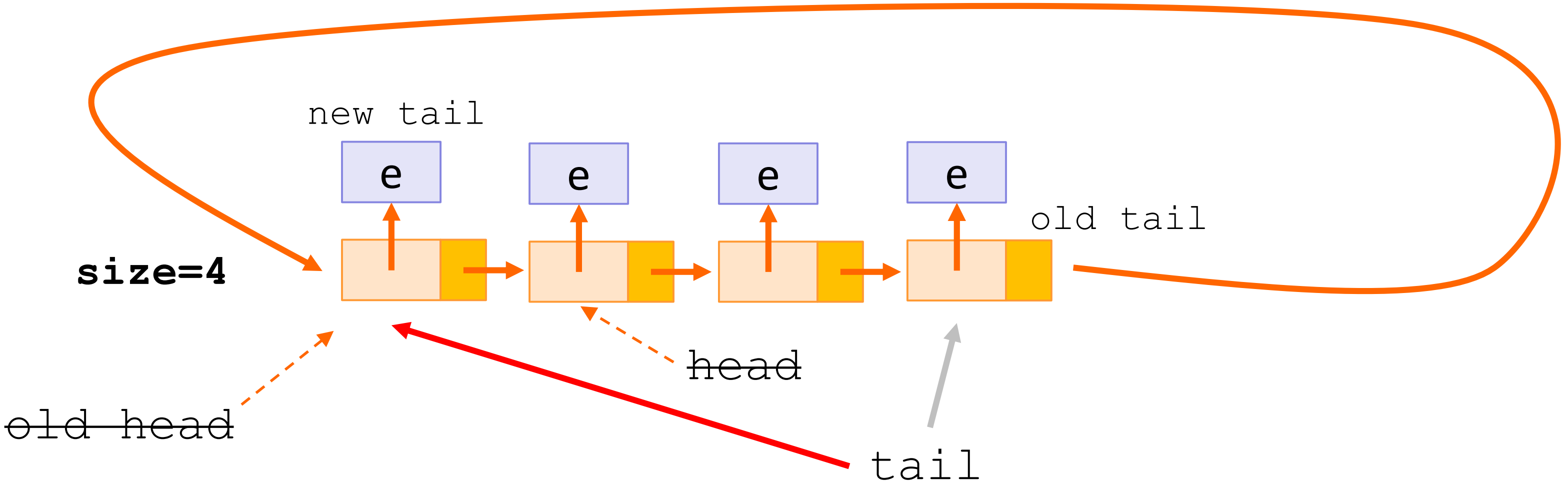
We can infer the head by `self._tail._next`

Circularly Linked List



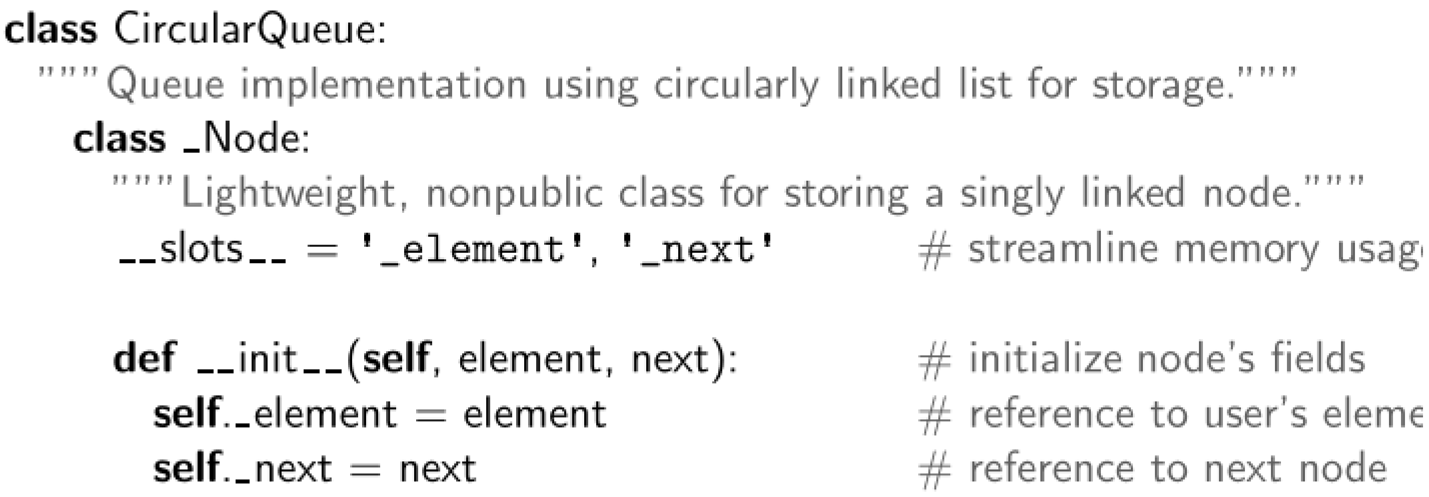
In case of enqueue operation, new node is inserted right after the tail and then the new node becomes the tail.

Circularly Linked List

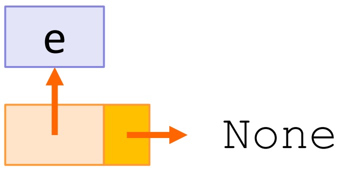


A new queue ADT operation, namely **`rotate`**, can be performed very efficiently. What `rotate` does is dequeue and item, process it, and then enqueue it. No need to `dequeue` and `enqueue` operations in practice, just process the item and move `tail` one step forward (make the head **new** `tail`) `self._tail = self._tail._next`

Circularly Linked List

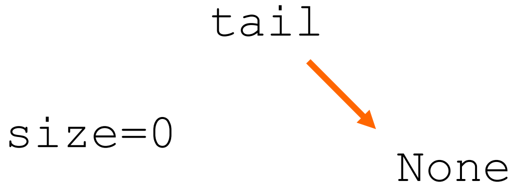


```
class CircularQueue:
  """Queue implementation using circularly linked list for storage."""
    class _Node:
      """Lightweight, nonpublic class for storing a singly linked node."""
      __slots__ = '_element', '_next'          # streamline memory usag

      def __init__(self, element, next):       # initialize node's fields
        self._element = element                # reference to user's eleme
        self._next = next                      # reference to next node
```



Circularly Linked List

```
def __init__(self):
  """Create an empty queue."""
  self._tail = None                    # will represent tail of 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
```



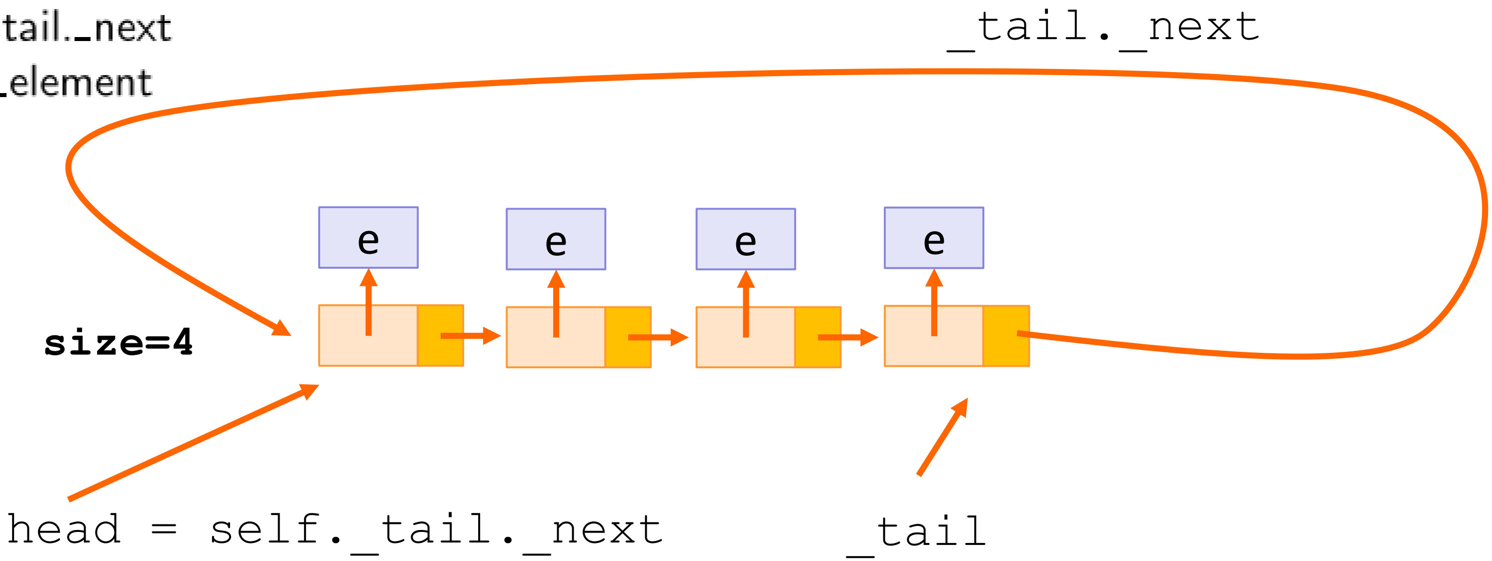
tail

size=0

None

Circularly Linked List

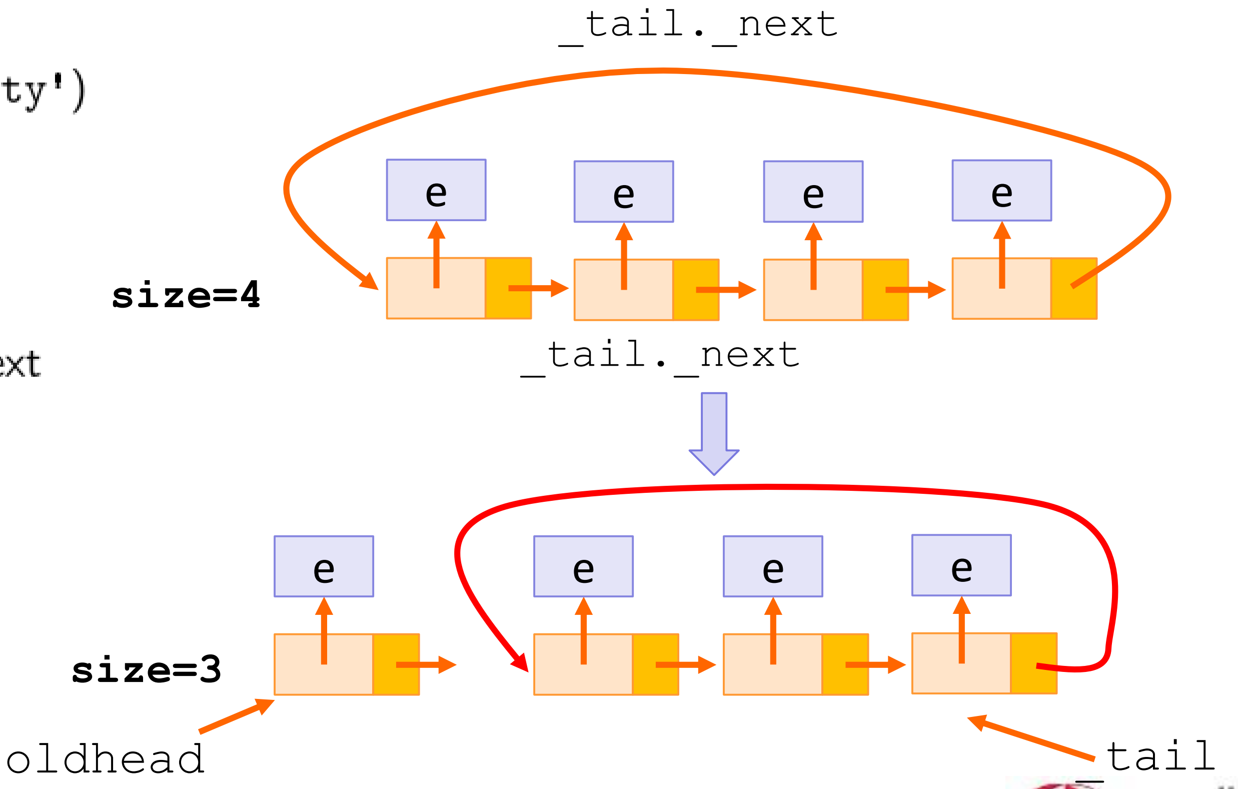
```
def first(self):
  """Return (but do not remove) the element at the front of the queue.

  Raise Empty exception if the queue is empty.
  """
  if self.is_empty():
    raise Empty('Queue is empty')
  head = self._tail._next
  return head._element
```



Circularly Linked List

```
def dequeue(self):
  """Remove and return the first element of the queue (i.e., FIFO).

  Raise Empty exception if the queue is empty.
  """
  if self.is_empty():
    raise Empty('Queue is empty')
  oldhead = self._tail._next
  if self._size == 1:
    self._tail = None
  else:
    self._tail._next = oldhead._next
  self._size −= 1
  return oldhead._element
```

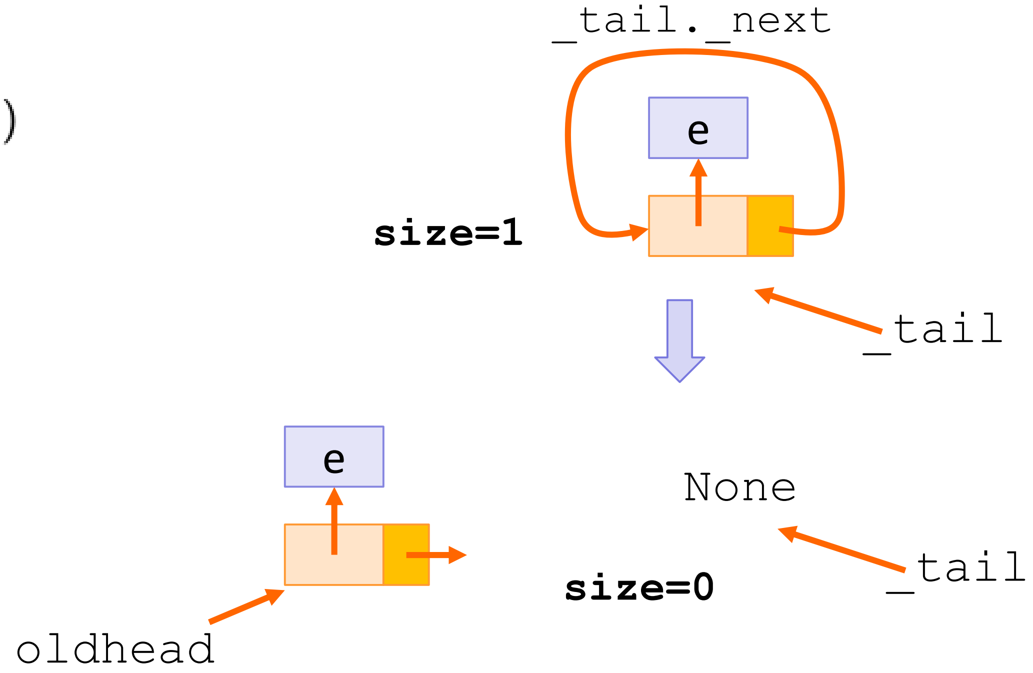
Case 1: Queue has more than 1 item



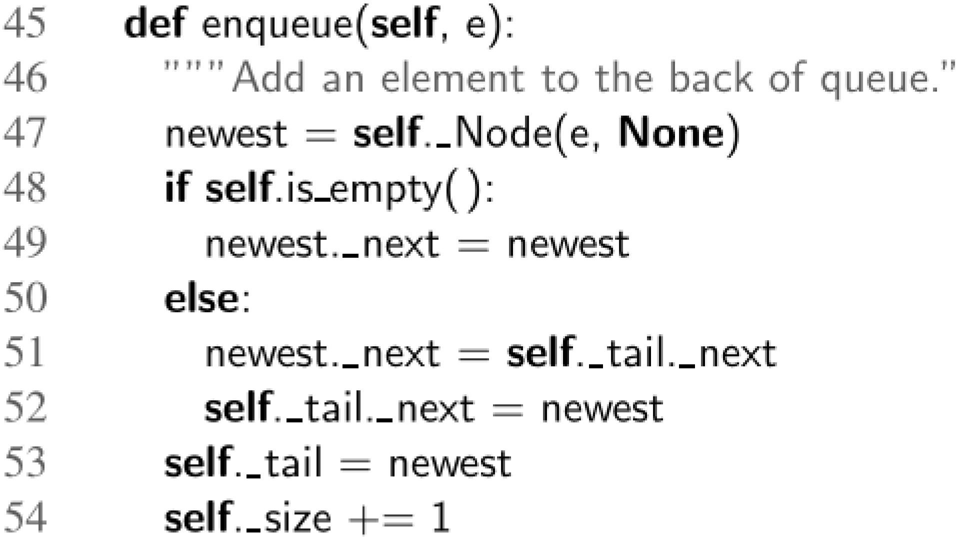
Circularly Linked List

```
def dequeue(self):
  """Remove and return the first element of the queue (i.e., FIFO).

  Raise Empty exception if the queue is empty.
  """
  if self.is_empty( ):
    raise Empty('Queue is empty')
  oldhead = self._tail._next
  if self._size == 1:
    self._tail = None
  else:
    self._tail._next = oldhead._next
  self._size −= 1
  return oldhead._element
```

Case 2: Queue has exactly 1 item

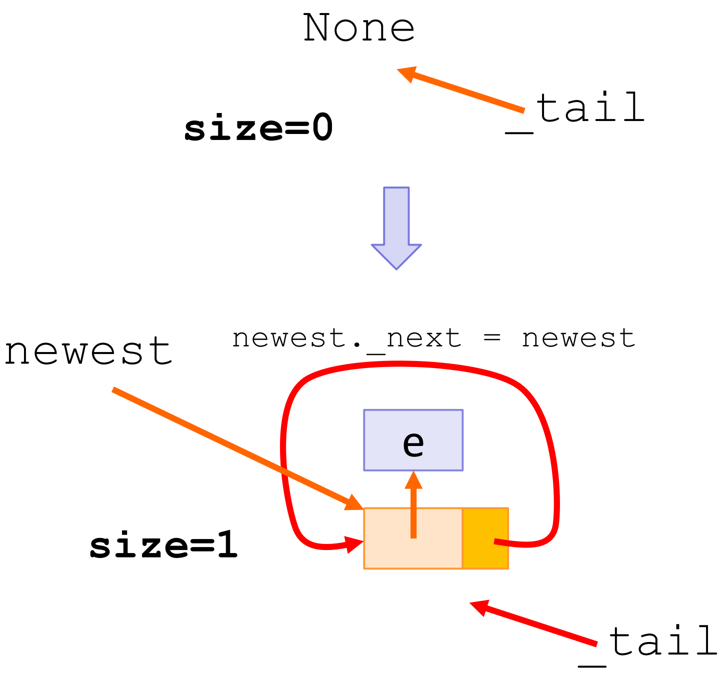


Circularly Linked List



```
def enqueue(self, e):
  """ Add an element to the back of queue."
  newest = self._Node(e, None)
  if self.is_empty( ):
    newest._next = newest
  else:
    newest._next = self._tail._next
    self._tail._next = newest
  self._tail = newest
  self._size += 1
```

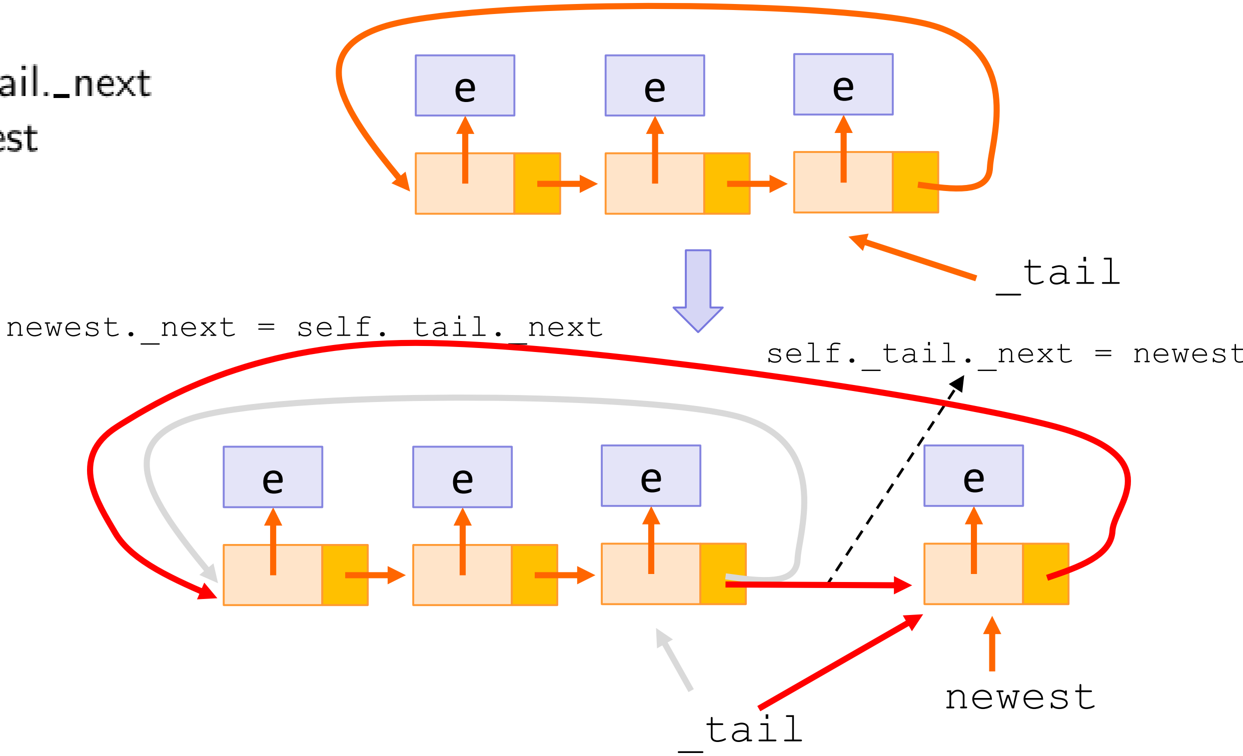
Case 1: Queue is empty



Circularly Linked List

```
def enqueue(self, e):
  """ Add an element to the back of queue."
  newest = self._Node(e, None)
  if self.is_empty():
    newest._next = newest
  else:
    newest._next = self._tail._next
    self._tail._next = newest
  self._tail = newest
  self._size += 1
```

Case 2: Queue is not empty



Doubly Linked List

Singly linked lists are asymmetric: Each node has a reference to the immediately following one.

This provides

- Efficient insertion at either end of the linked list
- Efficient deletion at the head of the list

However, it performs poorly when deleting from the tail.

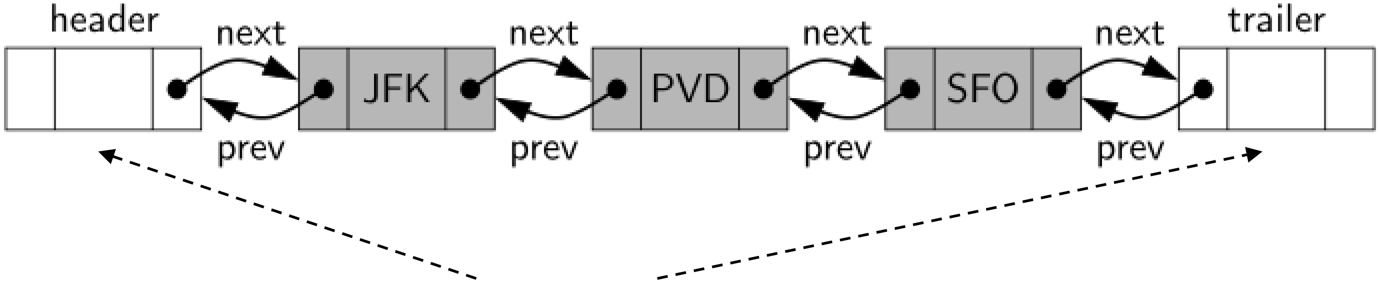
Doubly Linked List

With doubly linked lists,

Many different kinds of update operations,

Insertions and deletions at arbitrary positions

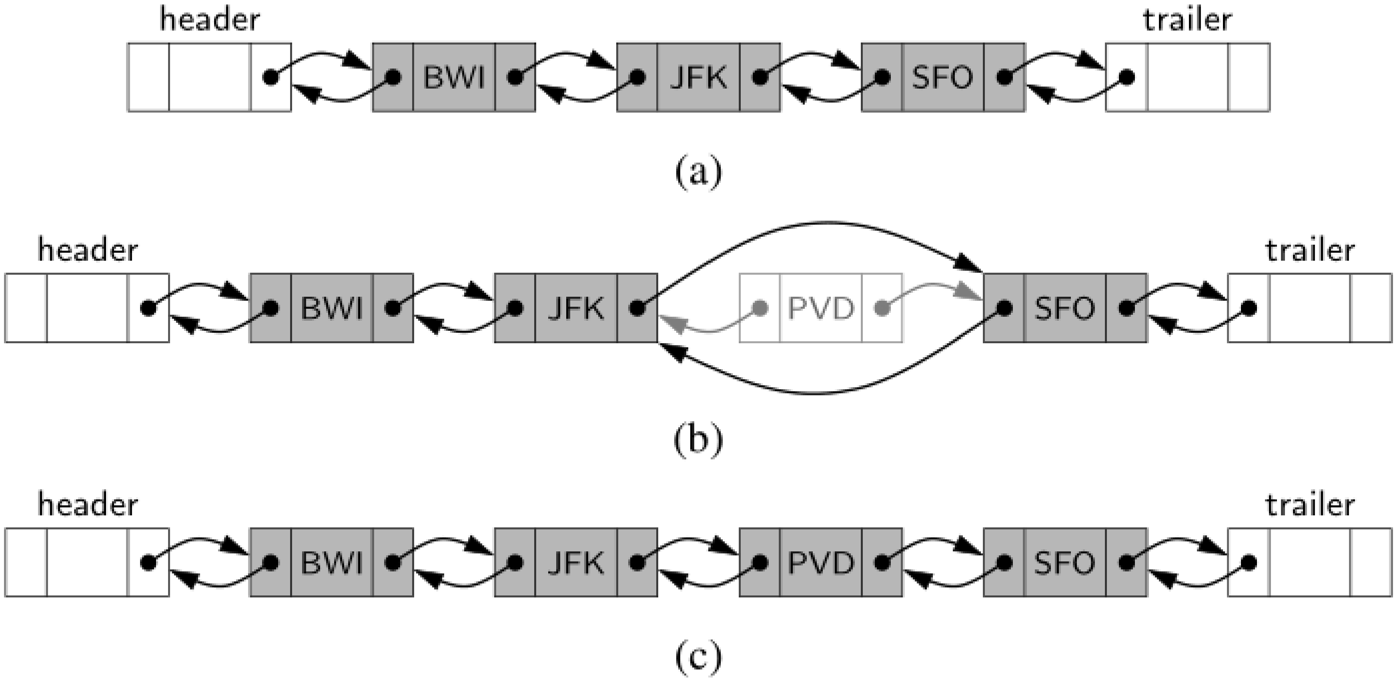
can be done efficiently.



These header and trailer nodes are called **sentinel nodes**. They do not store elements. They provide means to write more unified and simpler code for insertions and deletions.

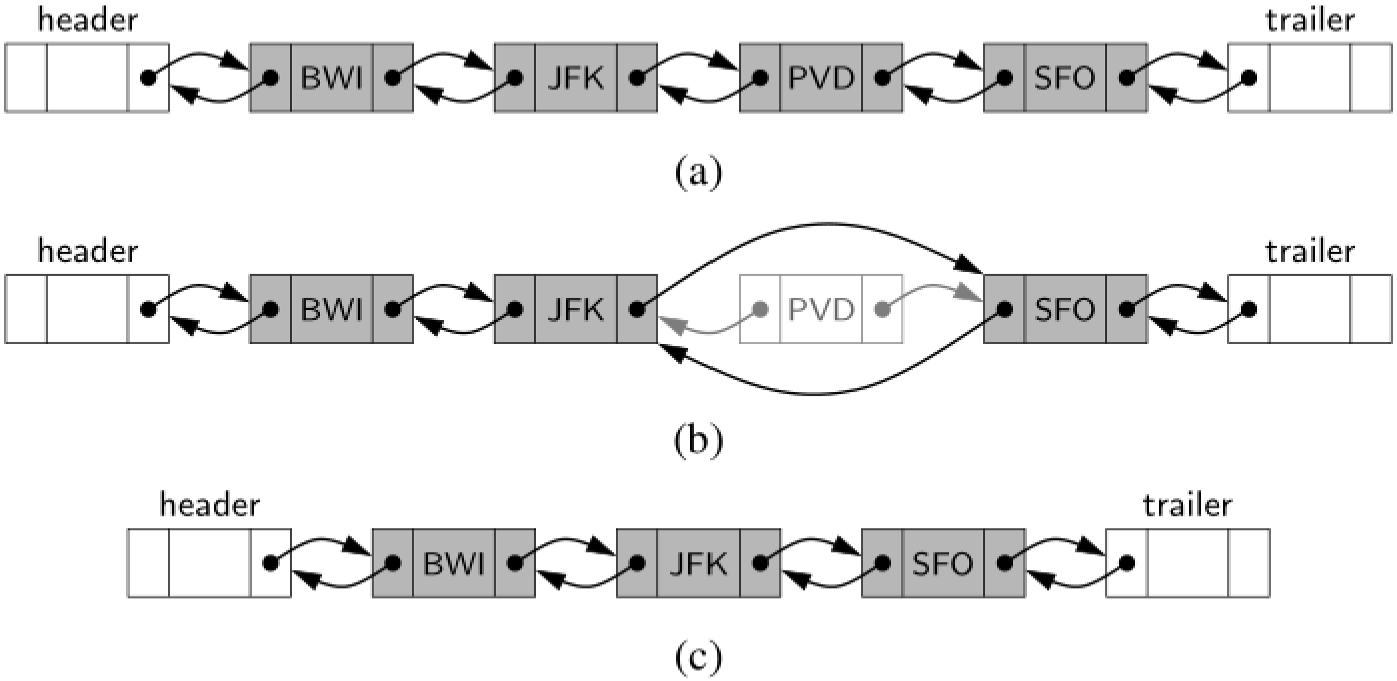
Doubly Linked List

Insertion Operation



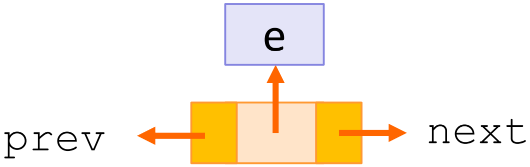
Doubly Linked List

Deletion Operation



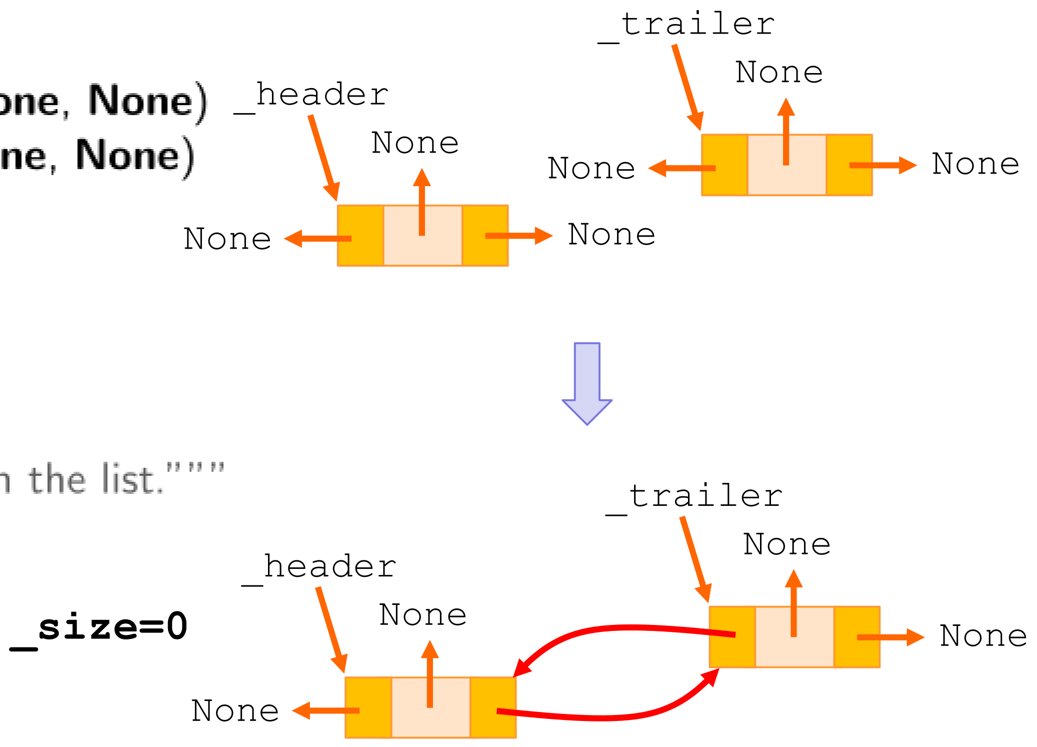
Doubly Linked List Implementation

```
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 fields
        self._element = element                    # user's element
        self._prev = prev                          # previous node referenc
        self._next = next                          # next node reference
```



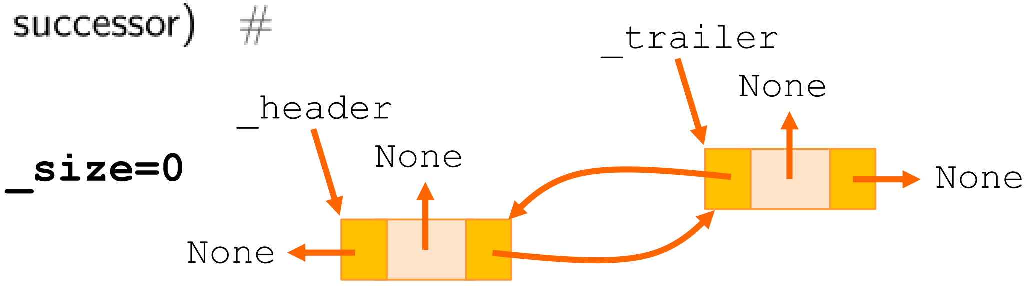
Doubly Linked List Implementation

```
  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
    self._trailer._prev = self._header
    self._size = 0

  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
```

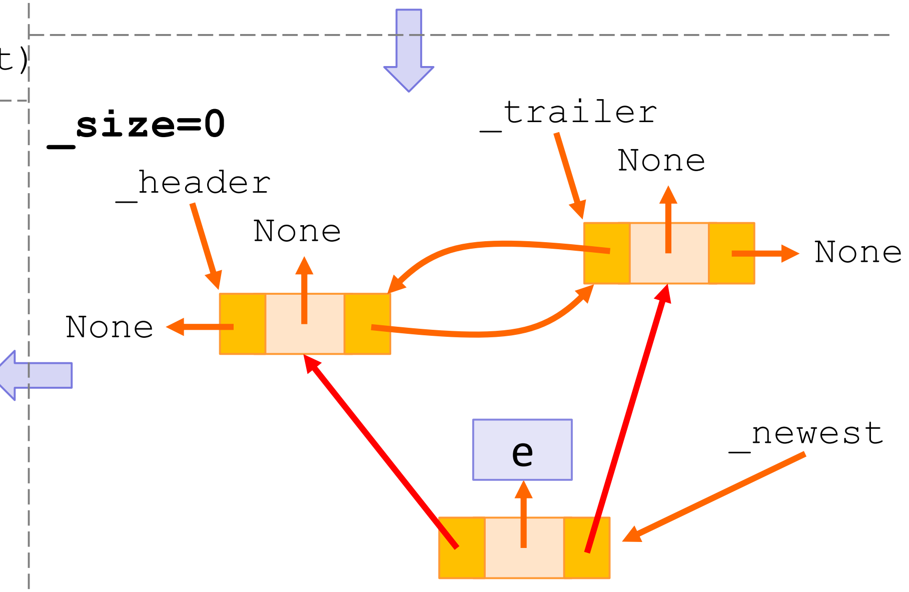


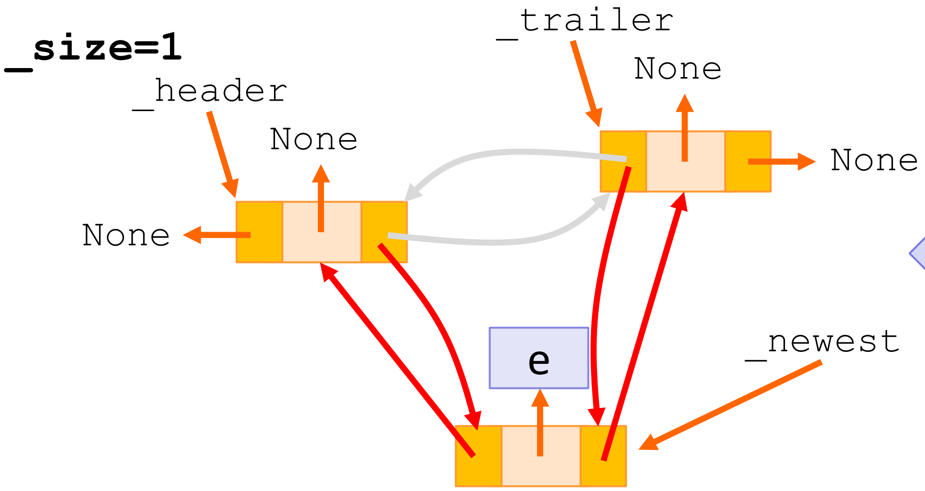
Doubly Linked List Implementation

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



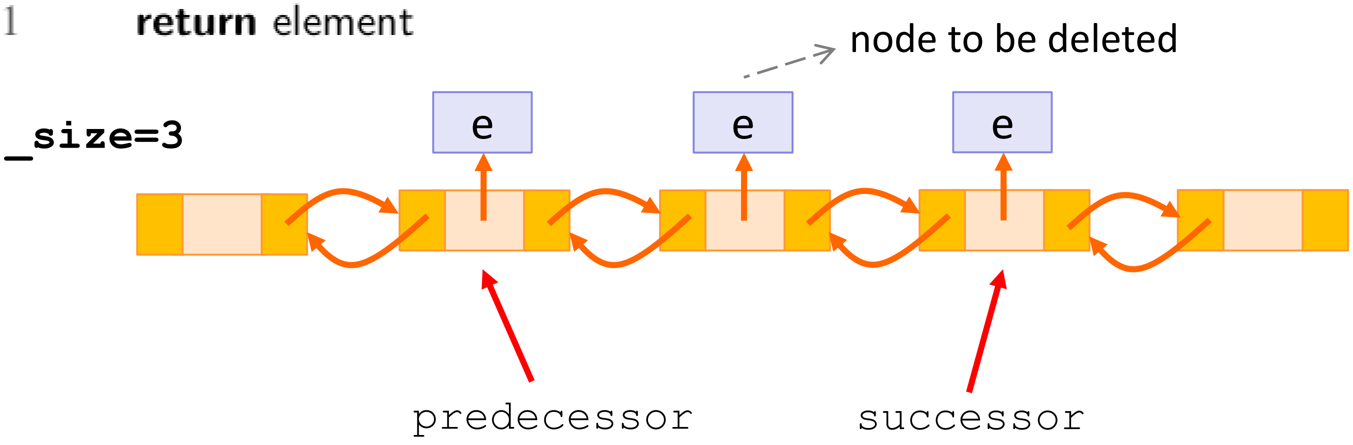
```
dl = _DoublyLinkBase()
dl._insert_between(e, dl._prev, dl._next)
```





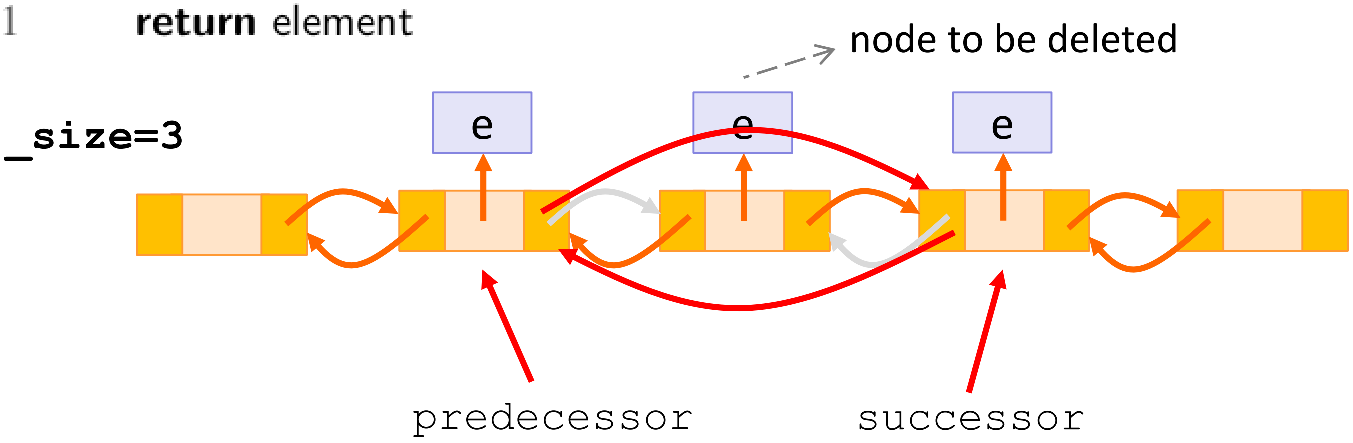
Doubly Linked List Implementation

```
def _delete_node(self, node):
  """Delete nonsentinel node from the list and return i
  predecessor = node._prev
  successor = node._next
  predecessor._next = successor
  successor._prev = predecessor
  self._size −= 1
  element = node._element
  node._prev = node._next = node._element = None
  return element
```



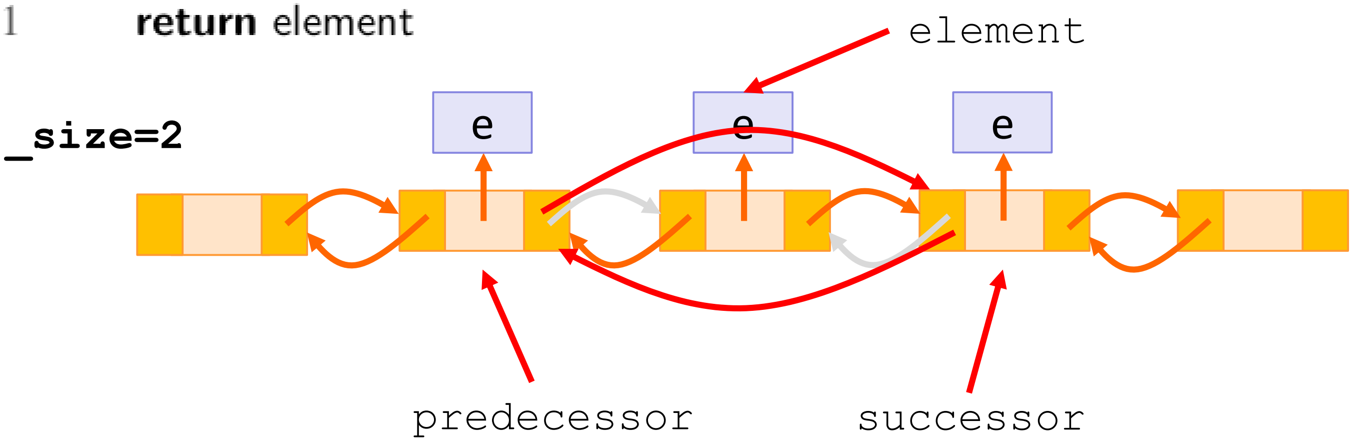
Doubly Linked List Implementation

```
def _delete_node(self, node):
    """Delete nonsentinel node from the list and return i
    predecessor = node._prev
    successor = node._next
    predecessor._next = successor
    successor._prev = predecessor
    self._size -= 1
    element = node._element
    node._prev = node._next = node._element = None
    return element
```



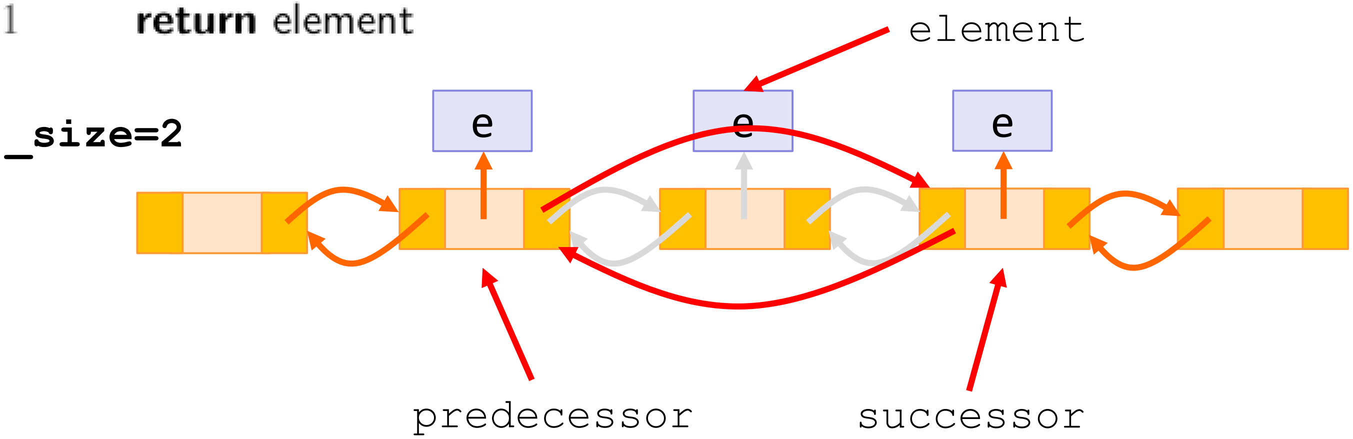
Doubly Linked List Implementation

```
def _delete_node(self, node):
  """Delete nonsentinel node from the list and return i
  predecessor = node._prev
  successor = node._next
  predecessor._next = successor
  successor._prev = predecessor
  self._size −= 1
  element = node._element
  node._prev = node._next = node._element = None
  return element
```



Doubly Linked List Implementation

```
def _delete_node(self, node):
  """Delete nonsentinel node from the list and return i
  predecessor = node._prev
  successor = node._next
  predecessor._next = successor
  successor._prev = predecessor
  self._size −= 1
  element = node._element
  node._prev = node._next = node._element = None
  return element
```



Link-based vs. Array-based Sequences

Element Access Complexity

Arrays provide O(1)-time access (memory address calculation with index is done in constant time)

In linked list, locating kth element requires O(k) time.

Link-based vs. Array-based Sequences

ADT Operation Costs

Almost all of the operations for arrays and linked lists are O(1), i.e., they have the same asymptotic upper bound.

However, arrays seem to be much better in terms of the number of CPU instructions that are performed.

- In arrays we just do simple math to find the necessary index, and very cheap assignment operations.
- In linked lists, we need to deal with internel objects (e.g., nodes), link assignment operations.

Link-based vs. Array-based Sequences

Memory Consumption

Array-based arrays tend to consume less memory than link-based arrays.

Both of them has references to the actual objects. So, in terms of the actual data, they consume the same amount of space.

In terms of the space spent for the references,

- Arrays spend, in the worst case, 2n reference spaces (right after a fresh reallocation, for appending n+1st item)
- Singly linked lists always spend at least 2n reference space, doubly linked lists spend 3n.

Link-based vs. Array-based Sequences

O(1) vs O(1) Amortized

Array-based structures' operations that require reallocation or deallocation (e.g., append) are amortized O(1).

All linked list operations are O(1) in the worst case.

For real time systems, structures that guarantee constant (the same) amount of time for "each" operation (i.e., O(1), rather than O(1) amortized) seem to be more suitable.

> For example, with an array, if we were to append n+1st item when the capacity is n, then that particular append operation would take n+1 unit time. Such delays may lead severe issues in real time systems.