Linked Lists

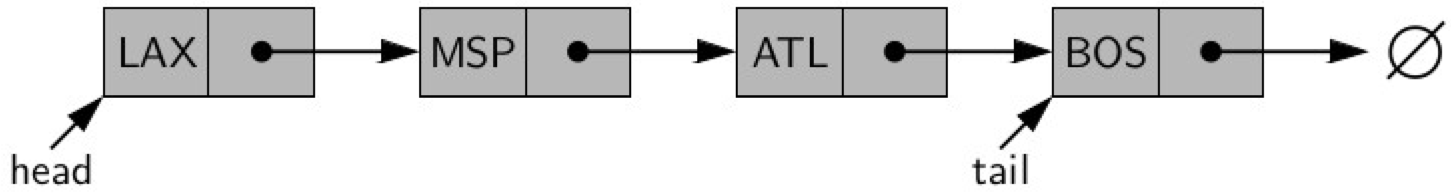
Limitations of array based implementations

1) The length is usually fixed apriori or if it is a dynamic array, it might be longer than the actual number of elements that it stores.

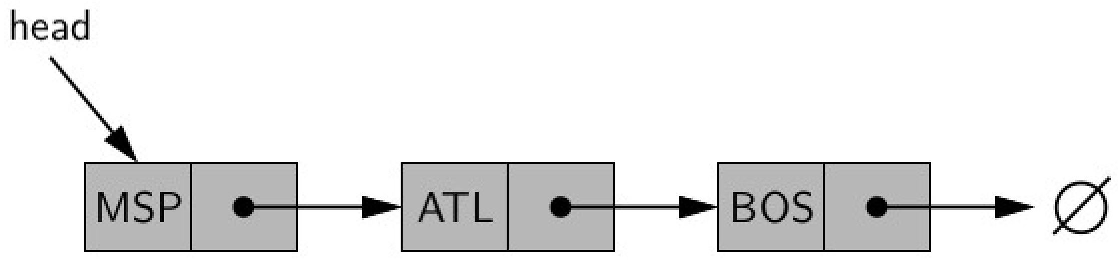
2) Amortized bounds for operations may be unacceptable in real-time systems

3) Insertions and deletions at interior positions of an array are expensive.

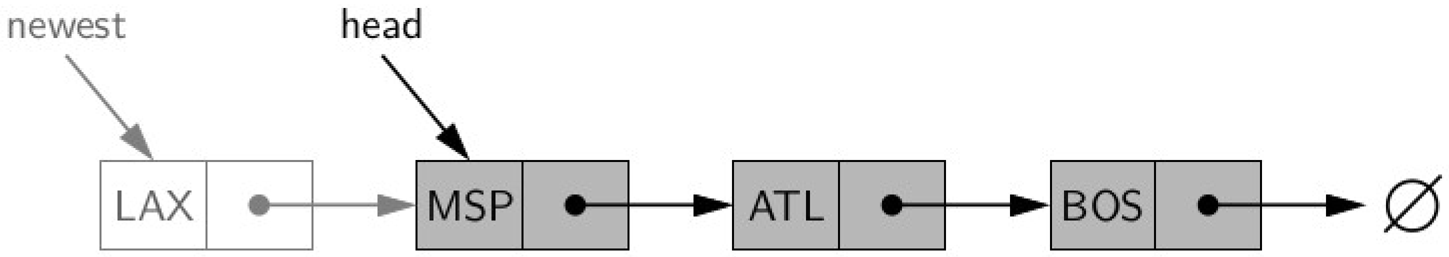
Singly Linked Lists



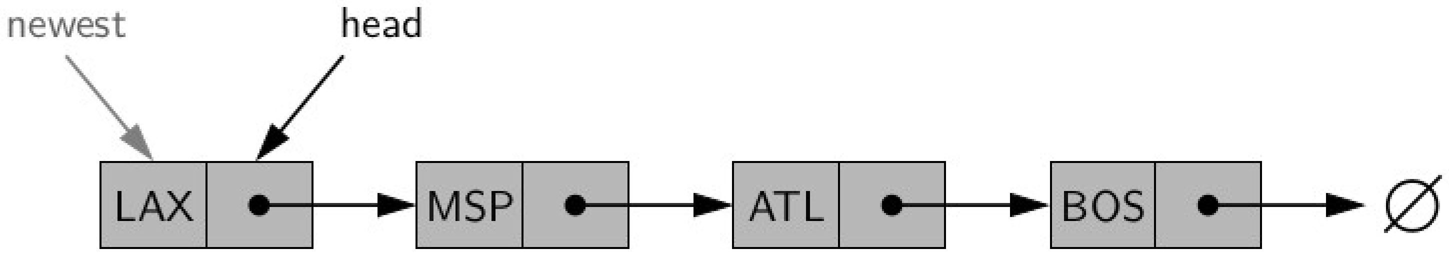
Inserting an Element at the Head of a Singly Linked List



(a)

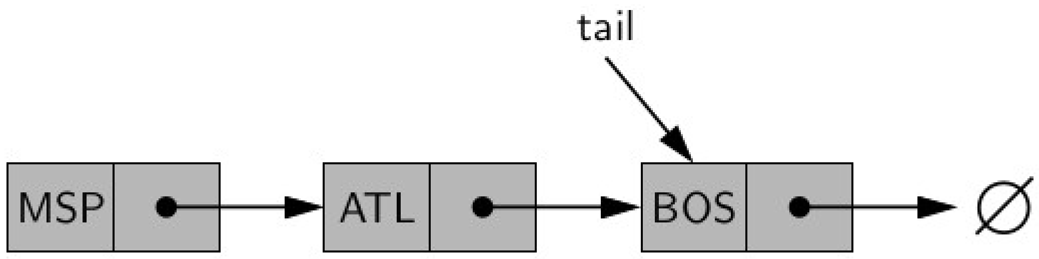


(b)

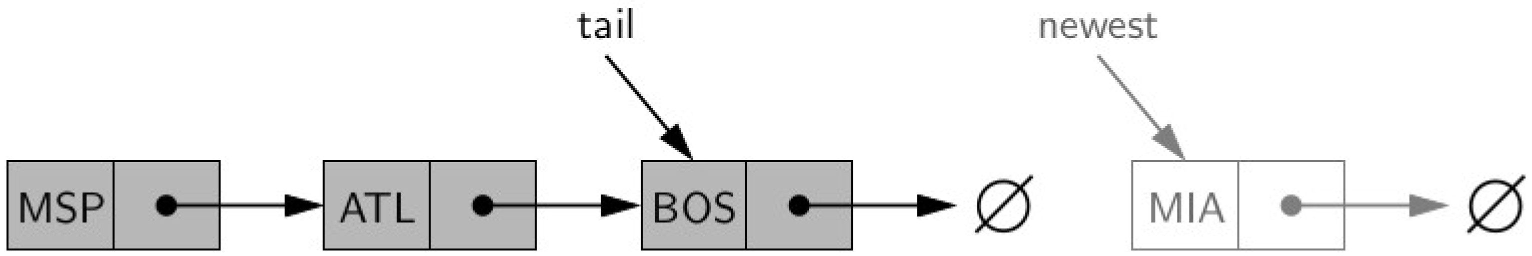


(c)

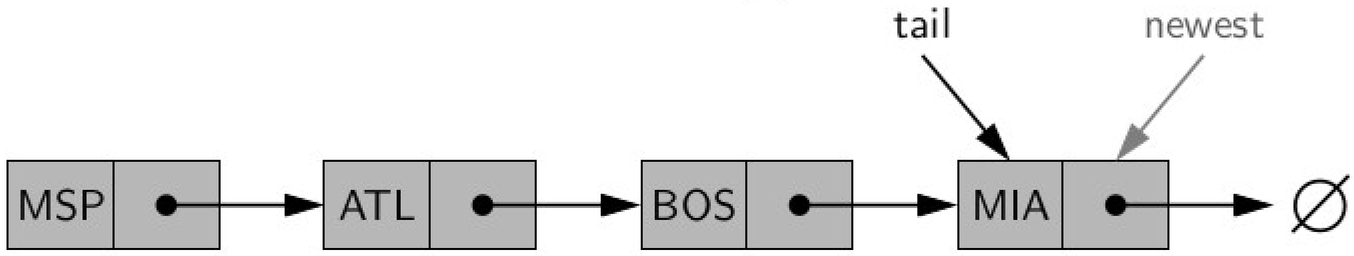
Inserting an Element at the Tail of a Singly Linked List



(a)

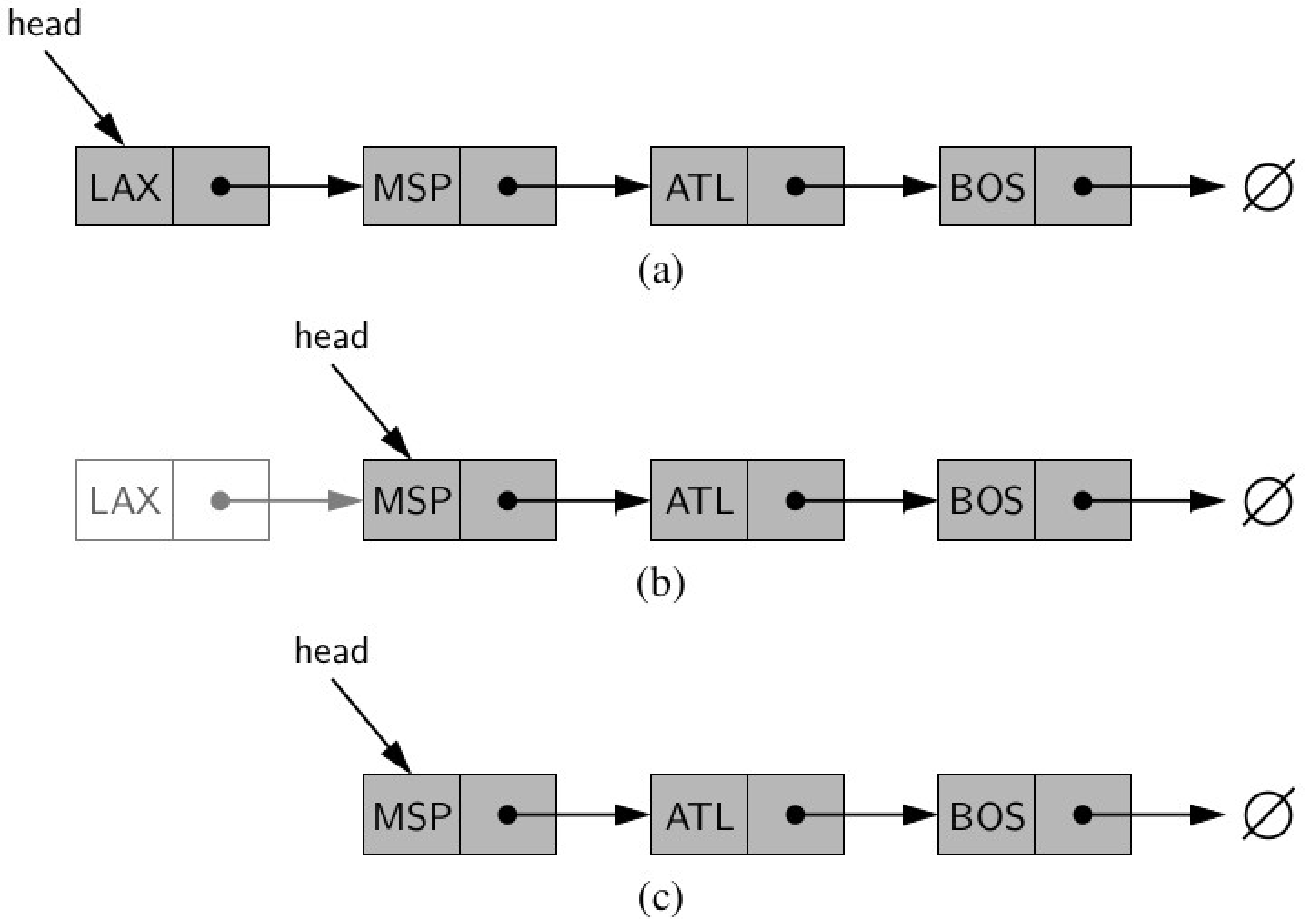


(b)



(c)

Removing an Element from a Singly Linked List



The 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
```

```
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

  #------------------------------- 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
```

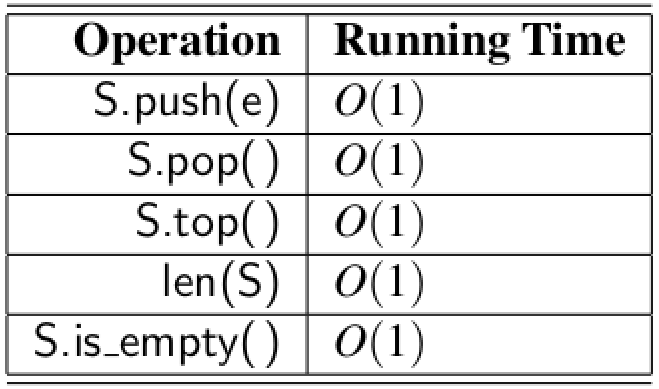
Stack Implementation using a Singly Linked List

```
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

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 stack is at head of list
```

Stack Implementation using a 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                # bypass the former top node
    self._size -= 1
    return answer
```

Performance of the LinkedStack implementation



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

All bounds are worst-case and the space usage is O(n), where n is the current number of elements in the stack.

Queue Implementation using a Singly Linked List

```
class LinkedQueue:
  """FIFO queue implementation using a singly linked list for storage."""

  class _Node:
    """Lightweight, nonpublic class for storing a singly linked node."""
    (omitted here; identical to that of LinkedStack._Node)

  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

```

Queue Implementation using a Singly Linked List

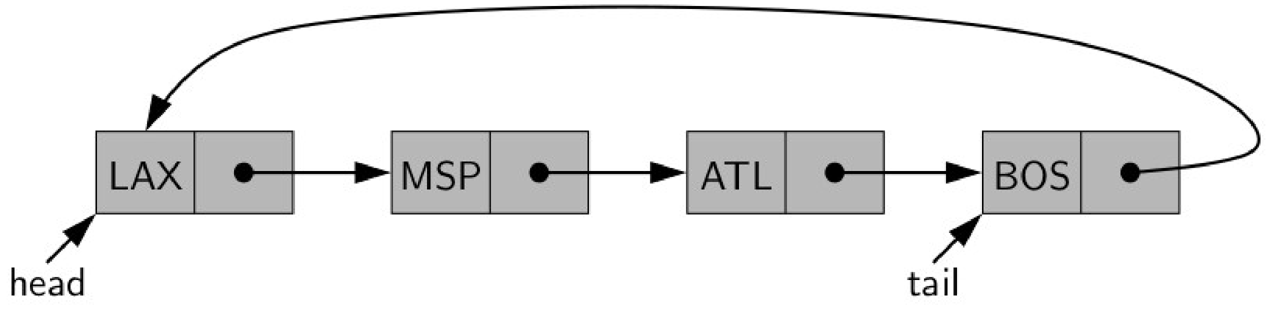
```

    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

    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():                         # special case as queue is empty
            self._tail = None                       # removed head had been the tail
        return answer
```

Queue Implementation using a Singly Linked List

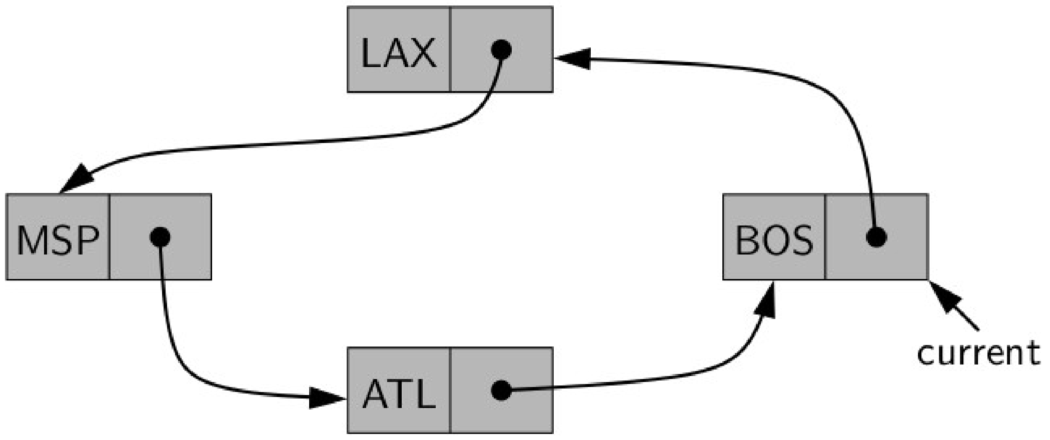
```
def enqueue(self, e):
  """Add an element to the back of queue."""
  newest = self._Node(e, None)          # node will be new tail node
  if self.is_empty():
    self._head = newest                 # special case: previously empty
  else:
    self._tail._next = newest
  self._tail = newest      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

Circularly Linked Lists

Example of a singly linked list with circular structure



Example of a circular linked list, with current denoting a reference to a select node.



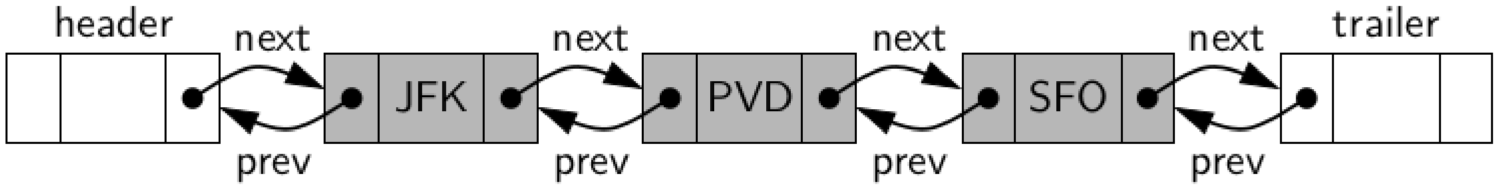
Circularly Linked Lists – key methods code walk-through

```
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

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:                      # removing only element
    self._tail = None                      # queue becomes empty
  else:
    self._tail._next = oldhead._next       # bypass the old head
  self._size -= 1
  return oldhead._element
```

Circularly Linked Lists – key methods code walk-through

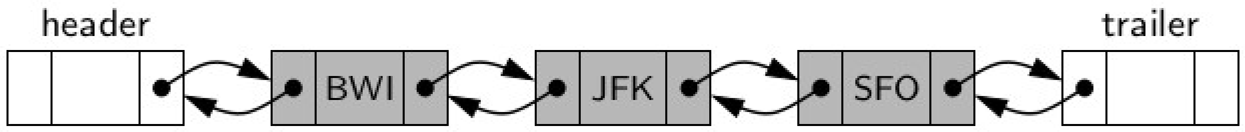
```
def enqueue(self, e):
  """Add an element to the back of queue."""
  newest = self._Node(e, None)              # node will be new tail node
  if self.is_empty():
    newest._next = newest                   # initialize circularly
  else:
    newest._next = self._tail._next         # new node points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

Doubly Linked Lists

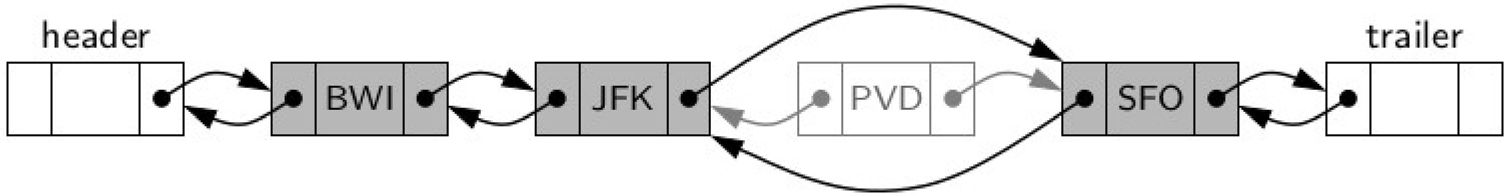


A doubly linked list representing the sequence { JFK, PVD, SFO }, using sentinels header and trailer to demarcate the ends of the list.

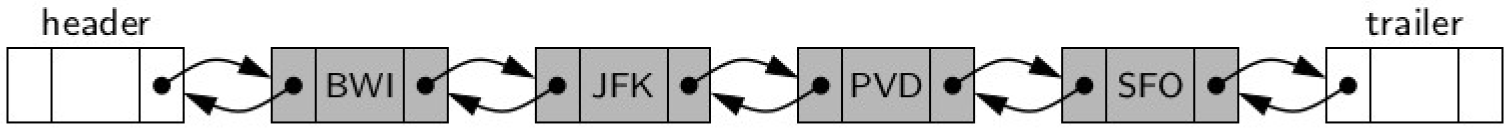
Adding an element to a doubly linked list (DLL) with header and trailer sentinels



(a)

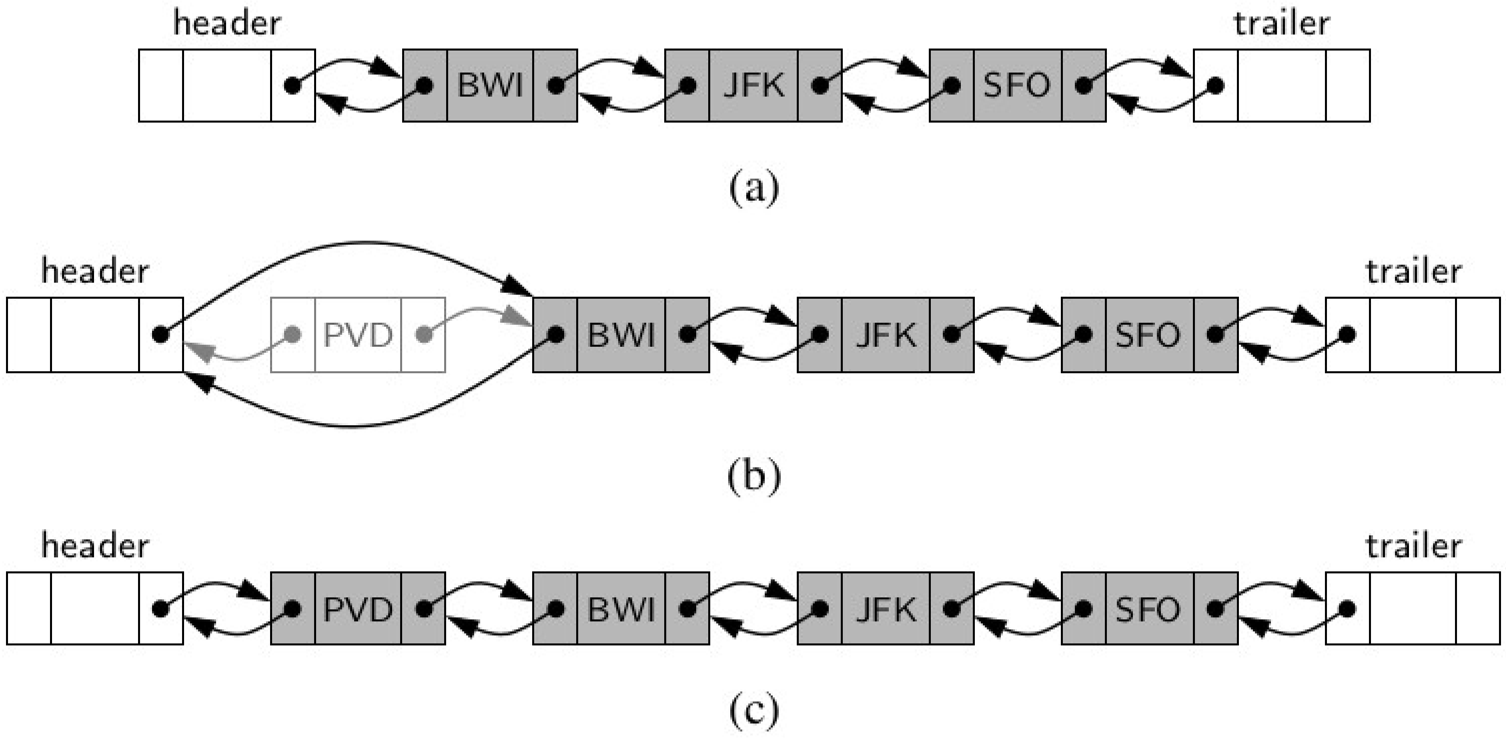


(b)

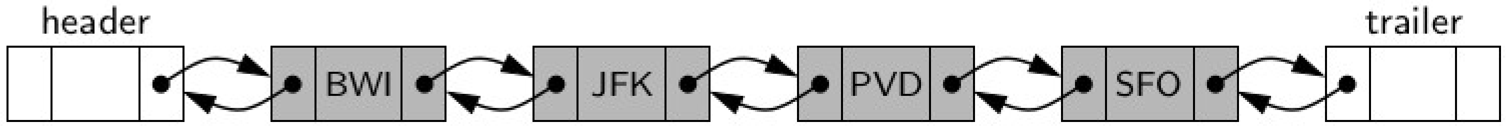


(c)

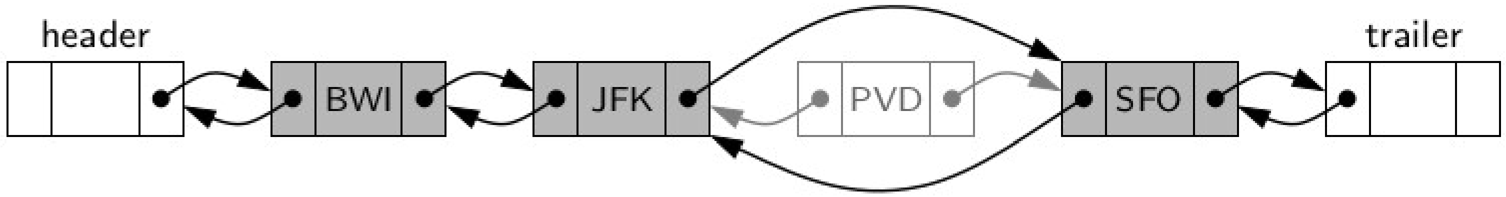
Adding an element to the front of a DLL



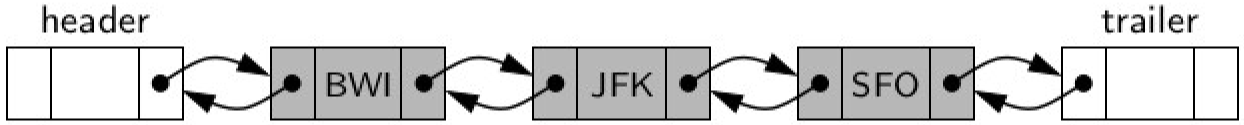
Deleting an element from a DLL



(a)



(b)



(c)

The Positional List Abstract Data Type – Basic Operations

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. See Section 1.8 for discussion of iterators in Python.

The Positional List Abstract Data Type – Basic Operations

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.

The Positional List Abstract Data Type – Basic Operations

A position instance is a simple object, supporting only the following method:

p.element( ): Return the element stored at position p.

Exercise problems

- Give an algorithm for finding the second-to-last node in a singly linked list in which the last node's next member points to a NULL.
- Describe a good algorithm for concatenating two singly linked lists L and M, given only references to the first node of each list, into a single list L that contains all the nodes of L followed by all the nodes of M.

Exercise problems

Suppose that x and y are references to nodes of circularly linked lists, although not necessarily the same list. Describe a fast algorithm for telling if x and y belong to the same list.

Our CircularQueue class of Section 7.2.2 provides a rotate( ) method that has semantics equivalent to Q.enqueue(Q.dequeue( )), for a nonempty queue. Implement such a method for the LinkedQueue class of Section 7.1.2 without the creation of any new nodes.

Exercise problems

Describe a nonrecursive method for finding, by link hopping, the middle node of a doubly linked list with header and trailer sentinels. In the case of an even number of nodes, report the node slightly left of center as the "middle." (Note: This method must only use link hopping; it cannot use a counter.) What is the running time of this method?

Exercise problems

Given the set of element {a, b, c, d, e, f } stored in a list, show the final state of the list, assuming we use the move-to-front heuristic and access the elements according to the following sequence: (a, b, c, d, e, f , a, c, f , b, d, e).

References

Data Structures and Algorithms in Python
Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser

Introduction to Algorithms
Leiserson,Stein,Rivest,Cormen

Algorithms, 4th Edition
Robert Sedgewick and Kevin Wayne

Few Images from the internet