香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen

Introduction to Computer Science: Programming Methodology

Lecture 10
Linked List

Prof. Pinjia He
School of Data Science

Why we need another list data type

- Python’s list class is highly optimized, and often a great choice for storage

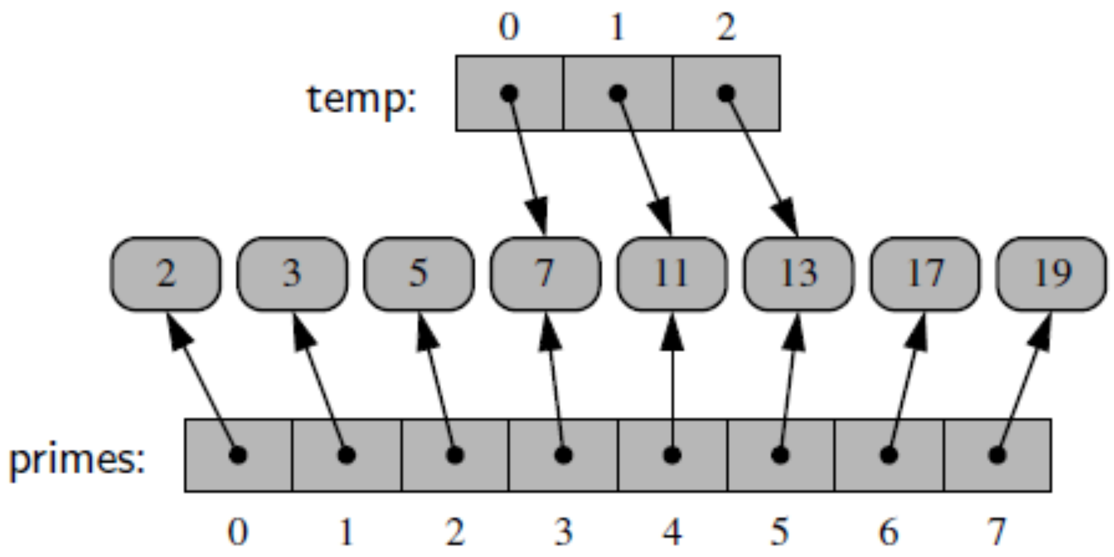
- However, many programming languages do not support this kind of optimized list data type

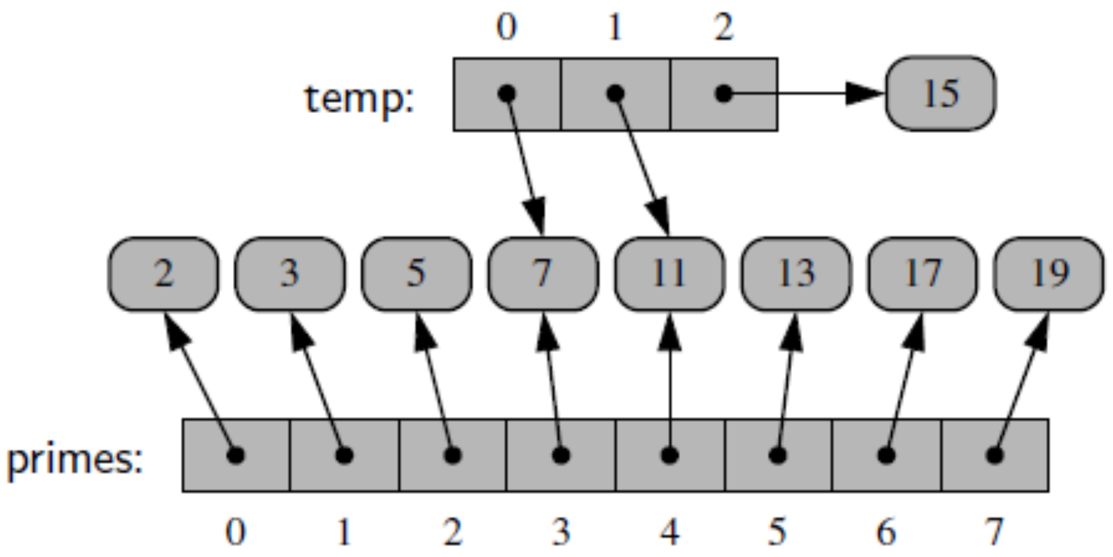
List in Python is a referential structure

```
>>> a=[1, 2, 3, 4, 5]
>>> for i in range(0, 5):
        print(id(a[i]))


1546964720
1546964752
1546964784
1546964816
1546964848
```

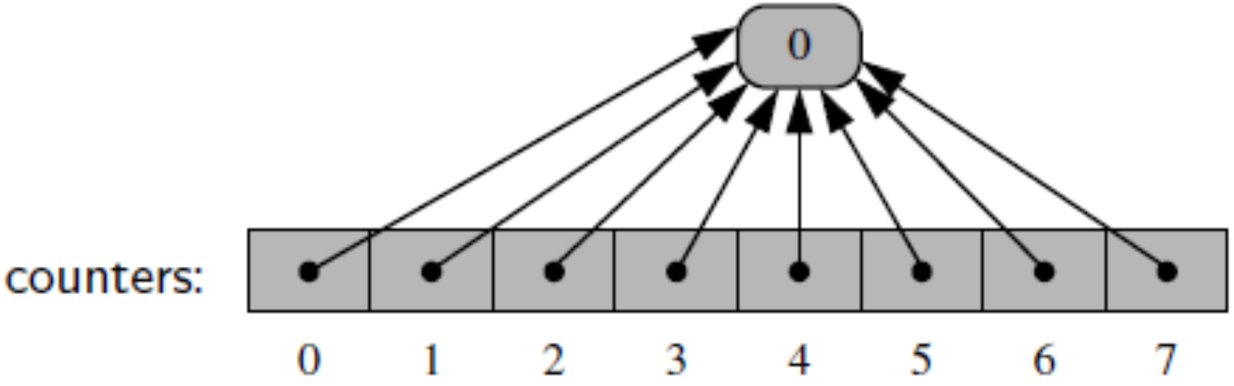
```
>>> a.insert(2, 10)
>>> a
[1, 2, 10, 3, 4, 5]
>>> for i in range(0, 6):
        print(id(a[i]))


1546964720
1546964752
1546965008
1546964784
1546964816
1546964848
```

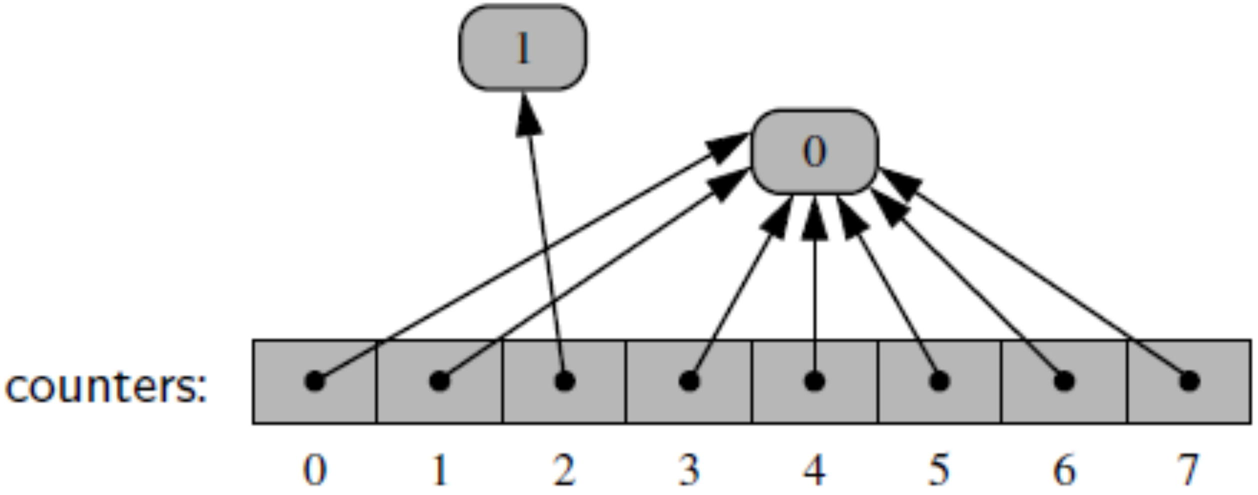
List in Python is a referential structure





List in Python is a referential structure





Compact array

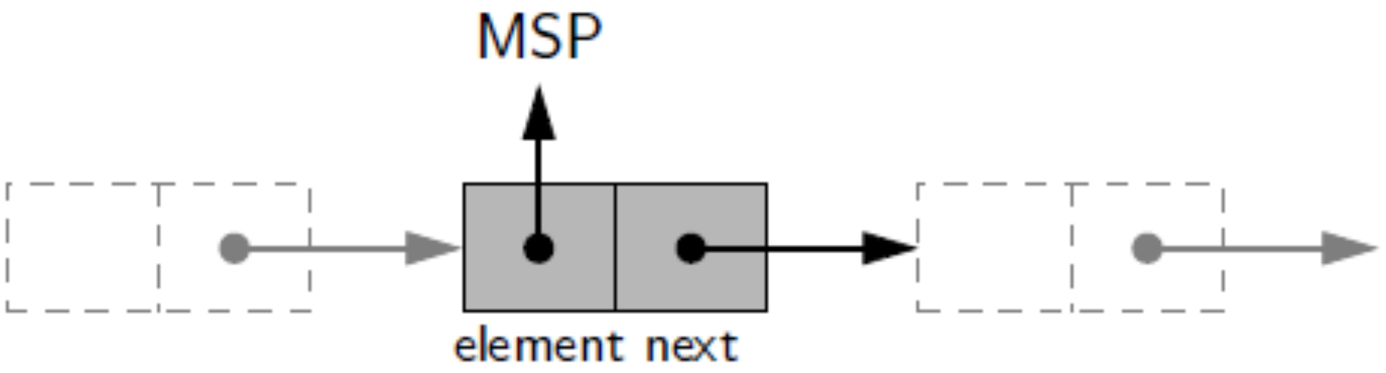
- A collection of numbers are usually stored as a compact array in languages such as C/C++ and Java

- A compact array is storing the bits that represent the primary data (not reference)

- The overall memory usage will be much lower for a compact structure because there is no overhead devoted to the explicit storage of the sequence of memory references (in addition to the primary data)

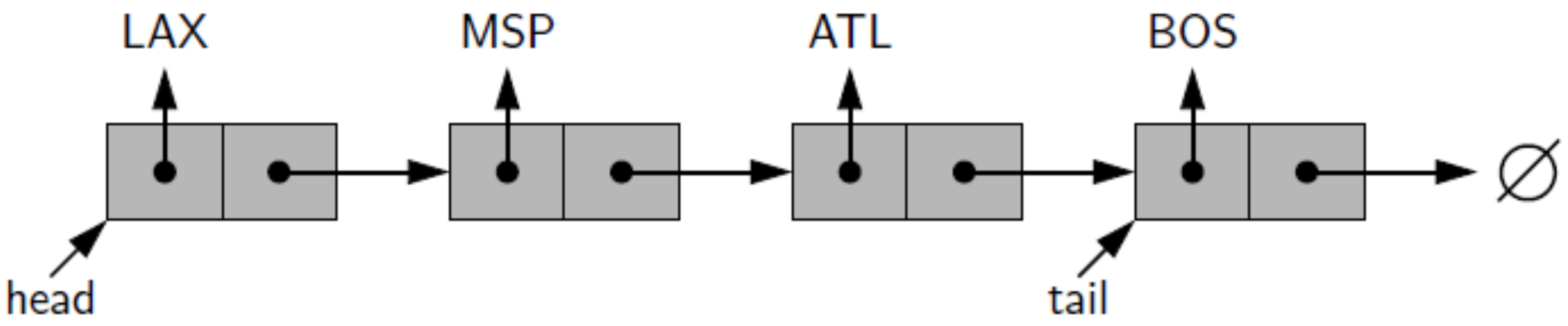
Linked List

- 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

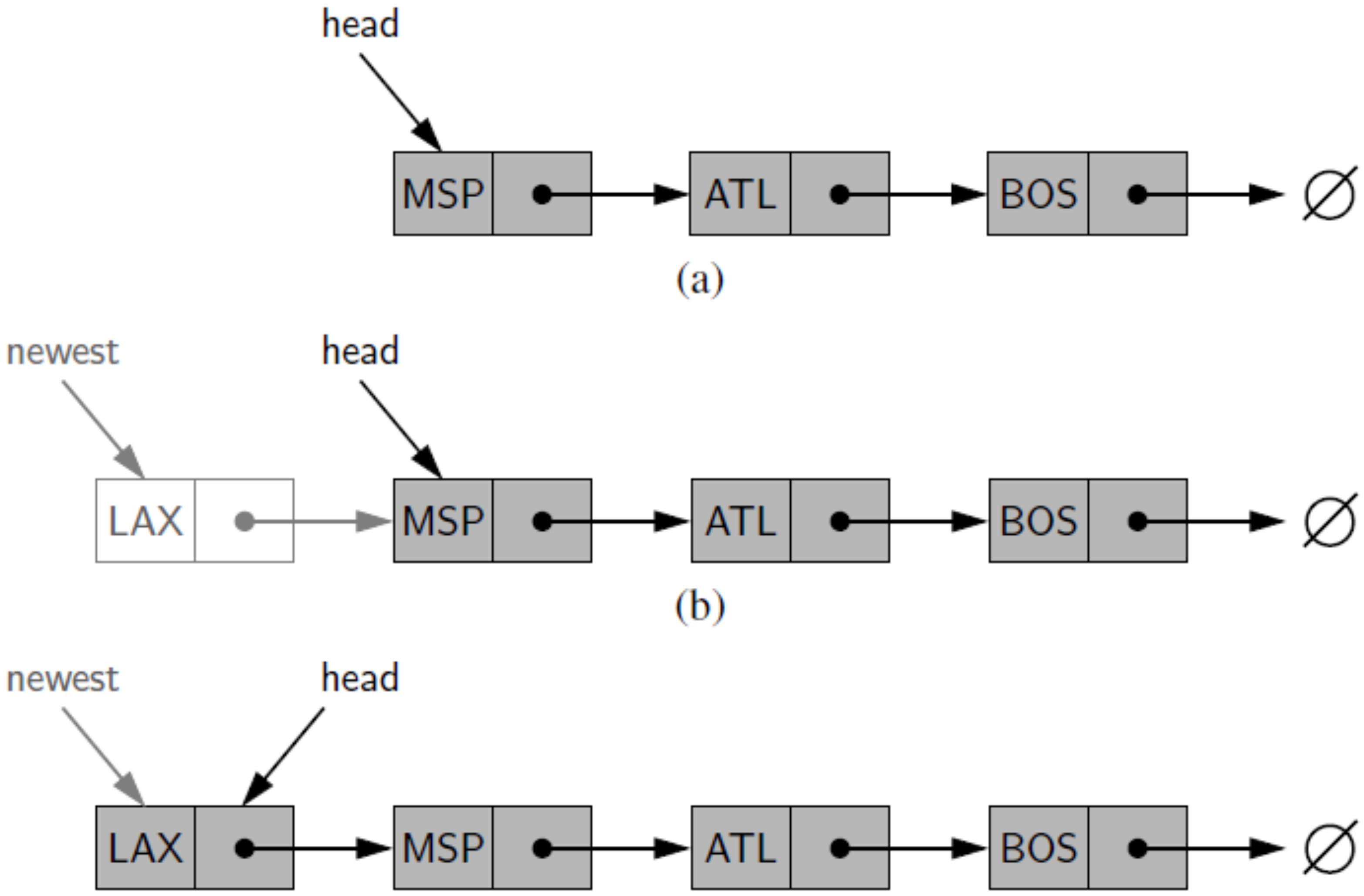


Linked List

- The first and last nodes 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
- We can identify the tail as the node having None as its next reference. This process is commonly known as traversing the linked list.
- 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



Inserting an Element at the Head of a Singly Linked List

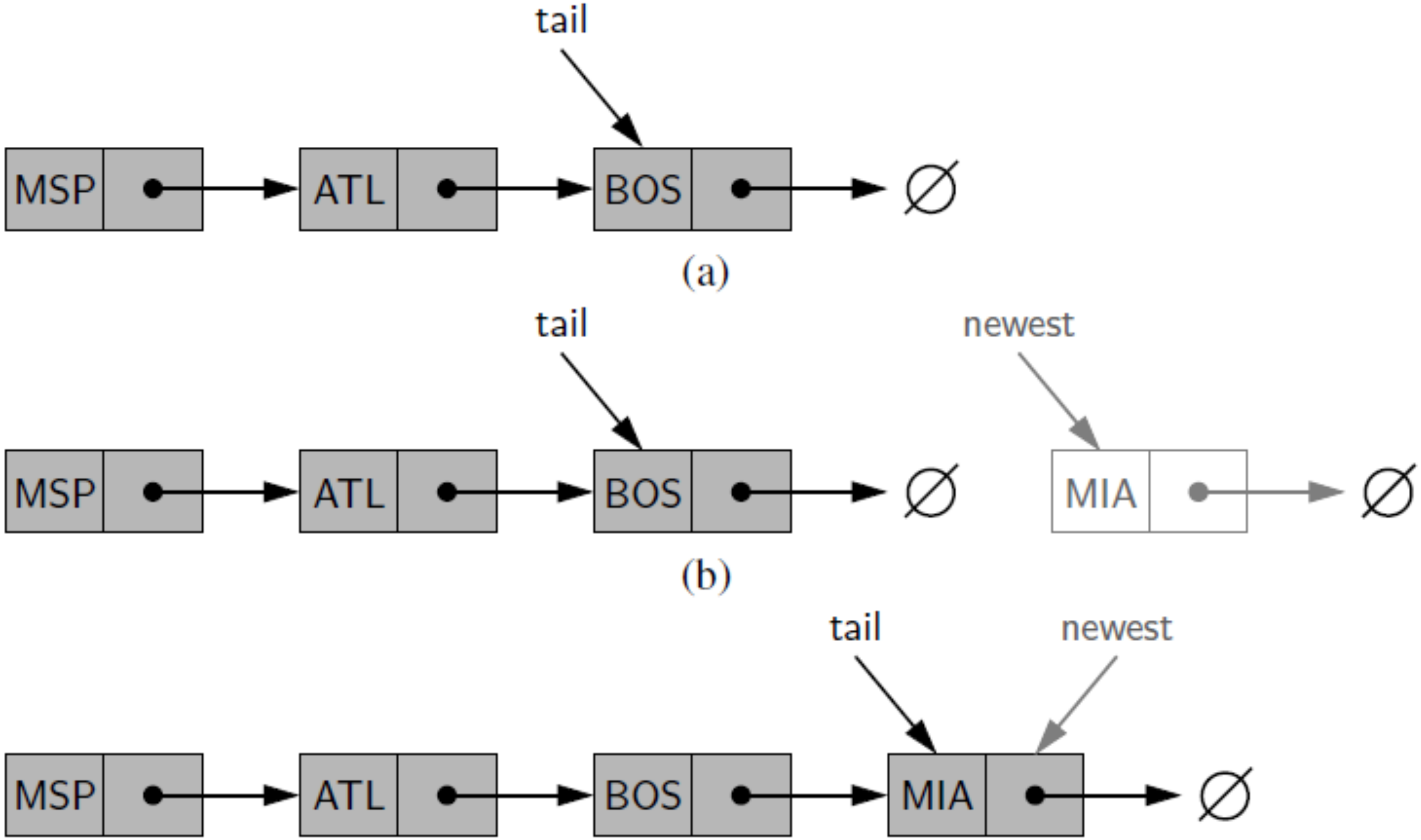


Pseudo code for inserting a node at the head

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

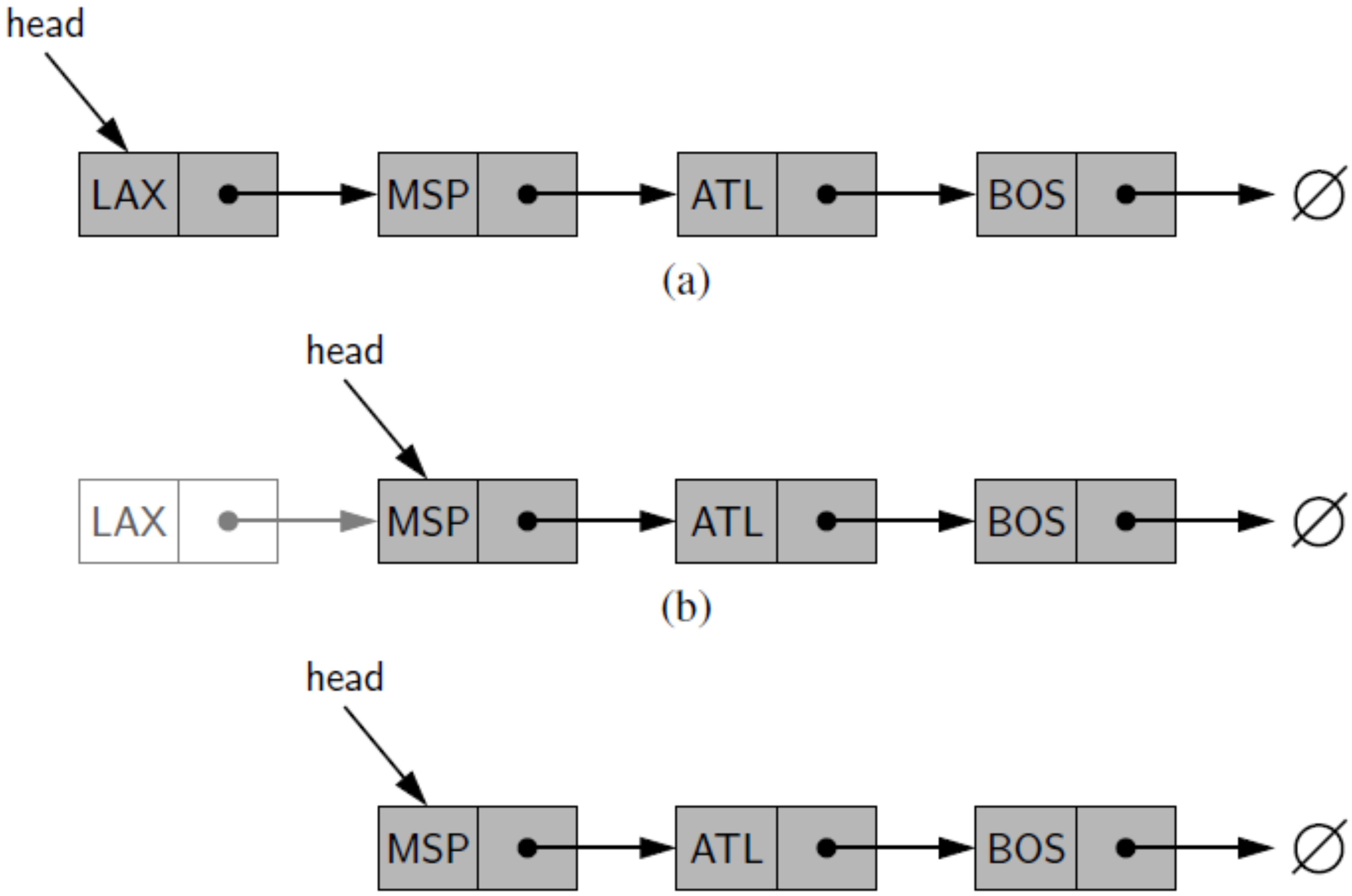


Pseudo code for inserting at the tail

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 the head of a Singly Linked List



Pseudo code for removing a node from the head

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}

Practice: Implement stack with a singly linked list

```
class Node:
    def __init__(self, element, pointer):
        self.element = element
        self.pointer = pointer

class LinkedStack:

    def __init__(self):
        self.head = None
        self.size = 0

    def __len__(self):
        return self.size

    def is_empty(self):
        return self.size == 0

    def push(self,e):
        self.head = Node(e, self.head)
        self.size += 1

    def top(self):
        if self.is_empty():
            print('Stack is empty.')
        else:
            return self.head.element

    def pop(self):
        if self.is_empty():
            print('Stack is empty.')
        else:
            answer = self.head.element
            self.head = self.head.pointer
            self.size -=1
            return answer
```

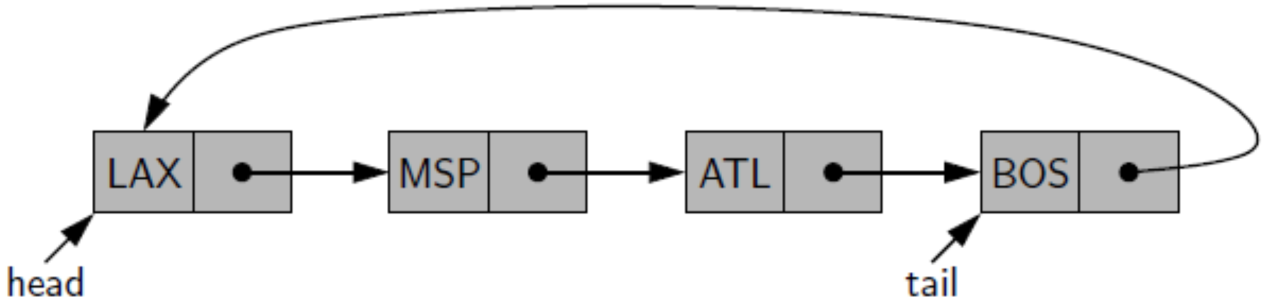
Practice: Implement queue with a singly linked list

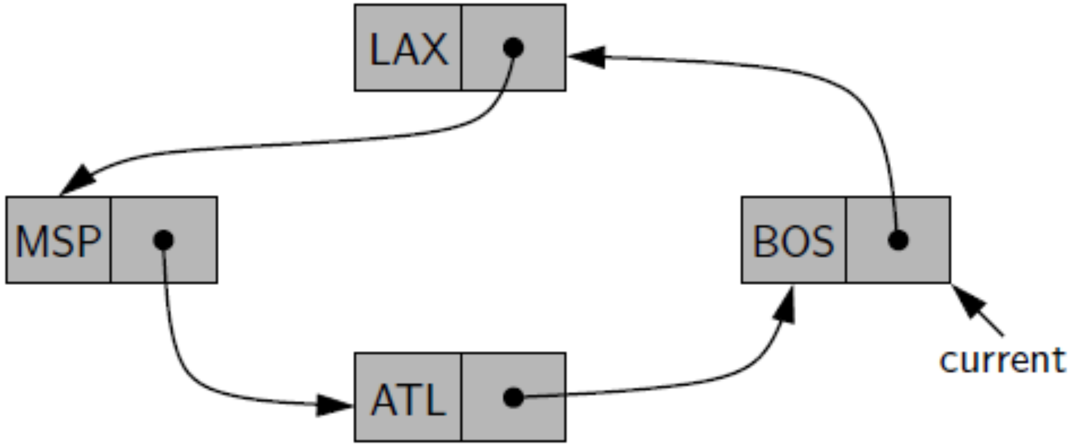
```
class LinkedQueue:

    def __init__(self):
        self. head = None
        self. tail = None
        self. size = 0

    def __len__(self):
        return self.size

    def is_empty(self):
        return self.size == 0

    def first(self):
        if self.is_empty():
            print('Queue is empty.')
        else:
            return self.head.element

    def dequeue(self):
        if self.is_empty():
            print('Queue is empty.')
        else:
            answer = self.head.element
            self.head = self.head.pointer
            self.size -= 1
            if self.is_empty():
                self.tail = None
            return answer

    def enqueue(self,e):
        newest = Node(e,None)

        if self.is_empty():
            self.head = newest
        else:
            self.tail.pointer = newest
        self.tail = newest
        self.size += 1
```

Circularly Linked List

- The tail of a linked list can use its next reference to point back to the head of the list

- Such a structure is usually called a circularly linked list





Example: Round-robin scheduler

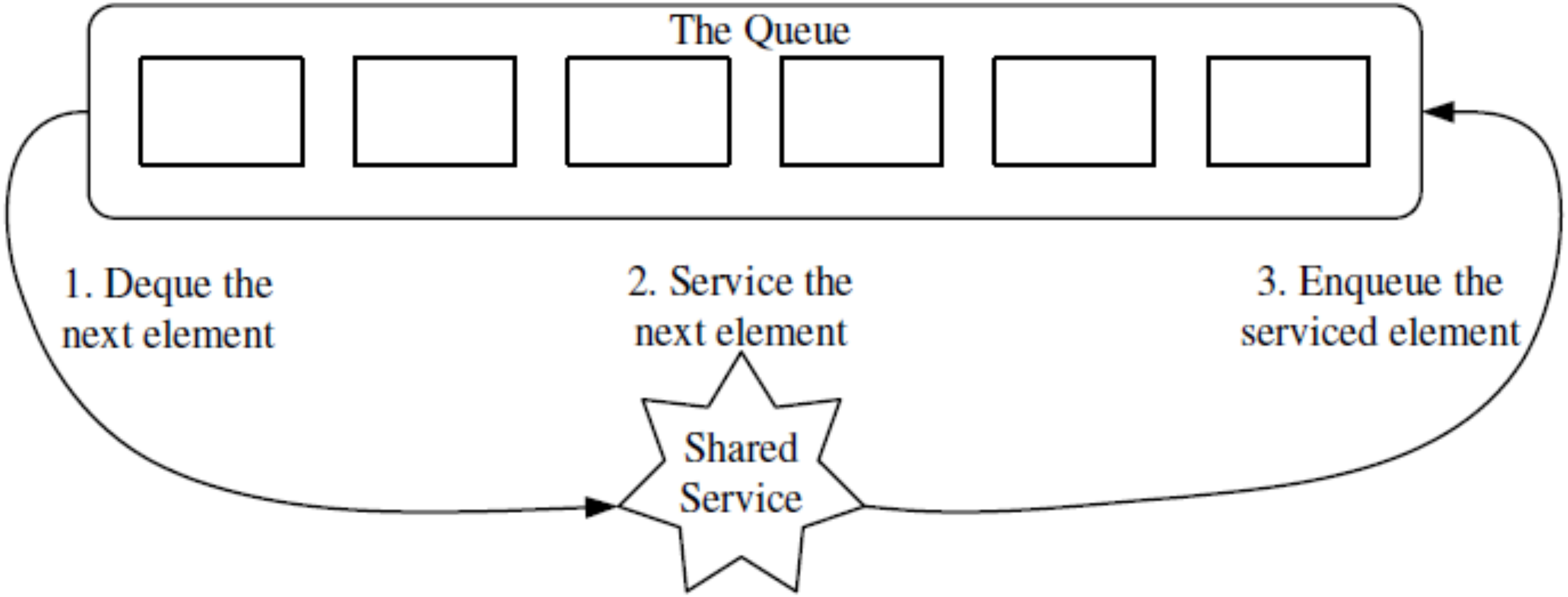
- A round-robin scheduler iterates through a collection of elements in a circular fashion and "serves" each element by performing a given action on it

- Such a scheduler is used, for example, to fairly allocate a resource that must be shared by a collection of clients

- For instance, round-robin scheduling is often used to allocate slices of CPU time to various applications running concurrently on a computer

Implementing round-robin scheduler using standard queue

- A round-robin scheduler could be implemented with the standard queue, by repeatedly performing the following steps on queue Q:
  1) e = Q.dequeue()
  2) Service element e
  3) Q.enqueue(e)



Implement a Queue with a Circularly Linked List

```
class Node:
    def __init__(self, element, pointer):
        self.element = element
        self.pointer = pointer

class CQueue:

    def __init__(self):
        self.__tail = None
        self.__size = 0

    def __len__(self):
        return self.__size

    def is_empty(self):
        return self.__size == 0

    def first(self):

        if self.is_empty():
            print('Queue is empty.')
        else:
            head = self.__tail.pointer
            return head.element
```

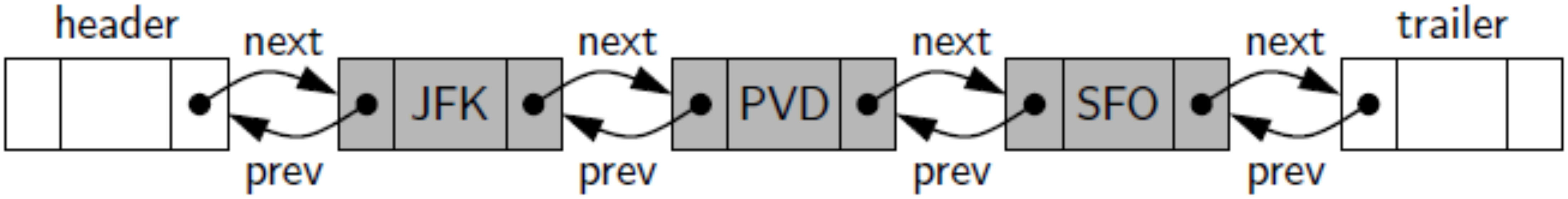
```python
def dequeue(self):
    if self.is_empty():
        print('Queue is empty.')
    else:
        oldhead = self.__tail.pointer
        if self.__size == 1:
            self.__tail = None
        else:
            self.__tail.pointer = oldhead.pointer
        self.__size -= 1
        return oldhead.element

def enqueue(self, e):
    newest = Node(e, None)
    if self.is_empty():
        newest.pointer = newest
    else:
        newest.pointer = self.__tail.pointer
        self.__tail.pointer = newest
    self.__tail = newest
    self.__size += 1
```

Doubly linked list

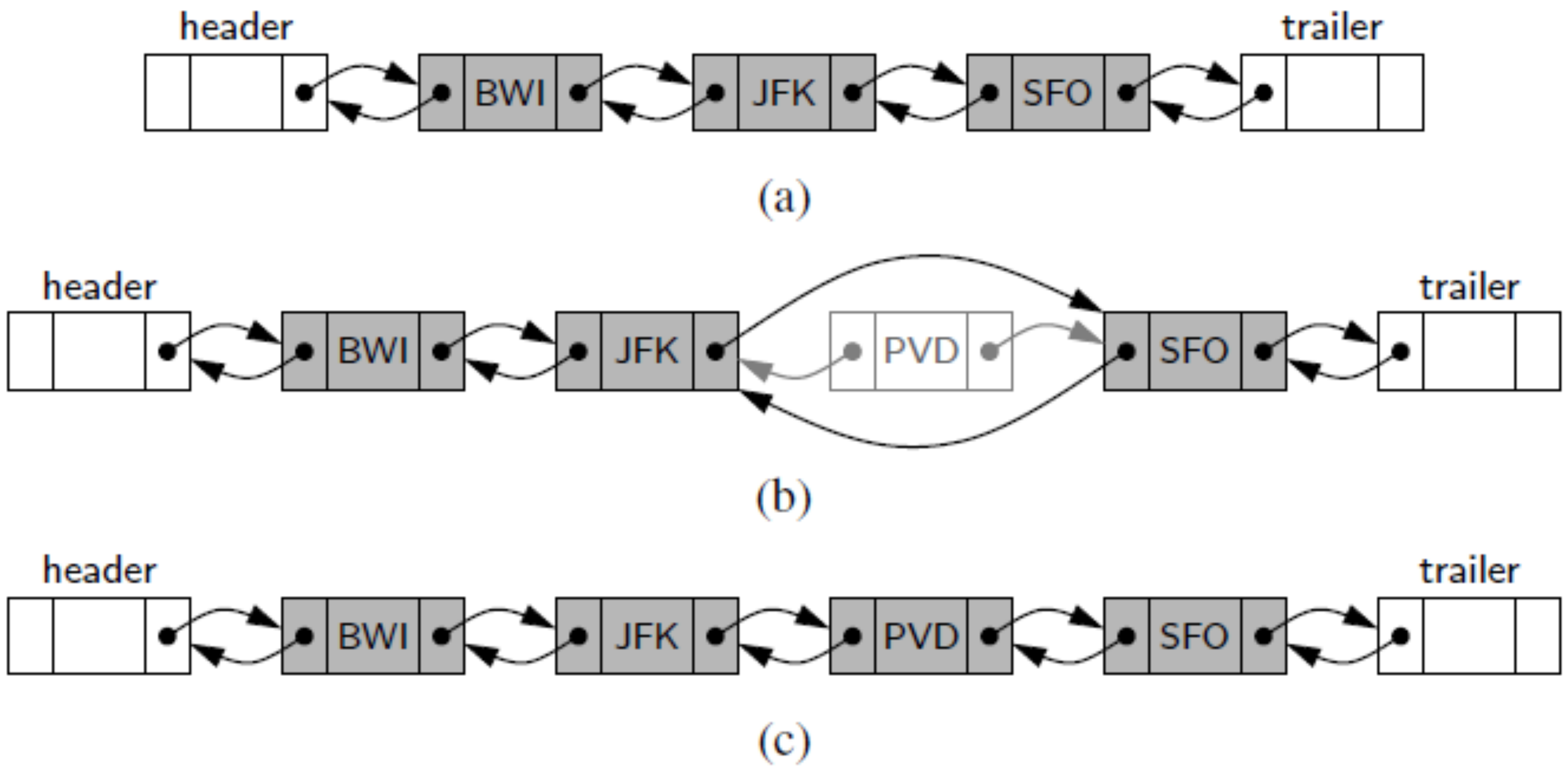
- For a singly linked list, we can efficiently insert a node at either end of a singly linked list, and can delete a node at the head of a list
- But we cannot efficiently delete a node at the tail of the list
- We can define a linked list in which each node keeps an explicit reference to the node before it and a reference to the node after it
- This kind of data structure is called doubly linked list

Head and tail sentinels

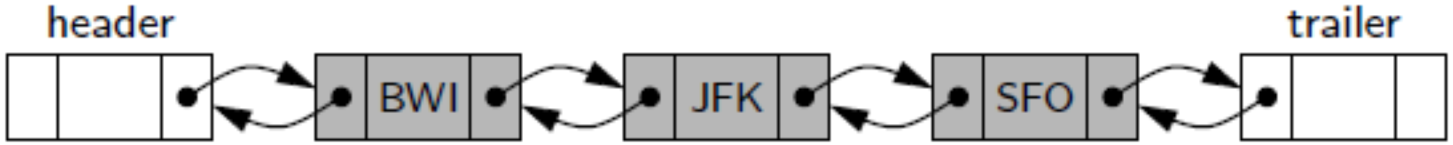
- In order to avoid some special cases when operating near the boundaries of a doubly linked list, it helps to add special nodes at both ends of the list: a header node at the beginning of the list, and a trailer node at the end of the list
- These "dummy" nodes are known as sentinels (or guards), and they do not store elements of the primary sequence



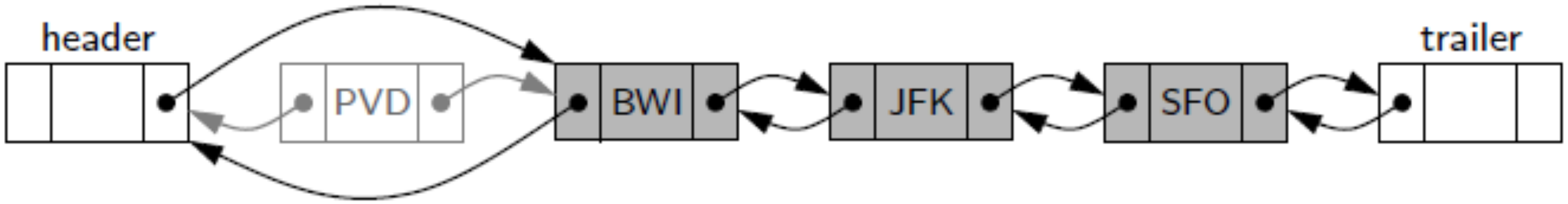
Inserting in the middle of a doubly linked list



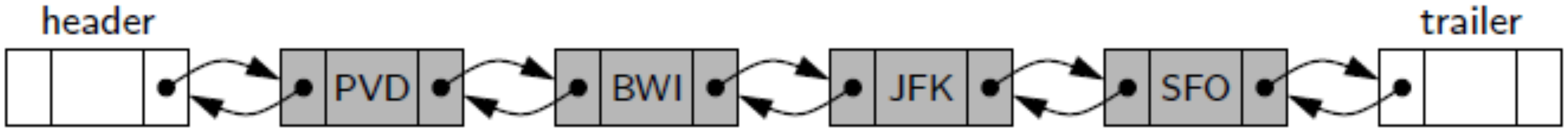
Inserting at the head of the doubly linked list



(a)

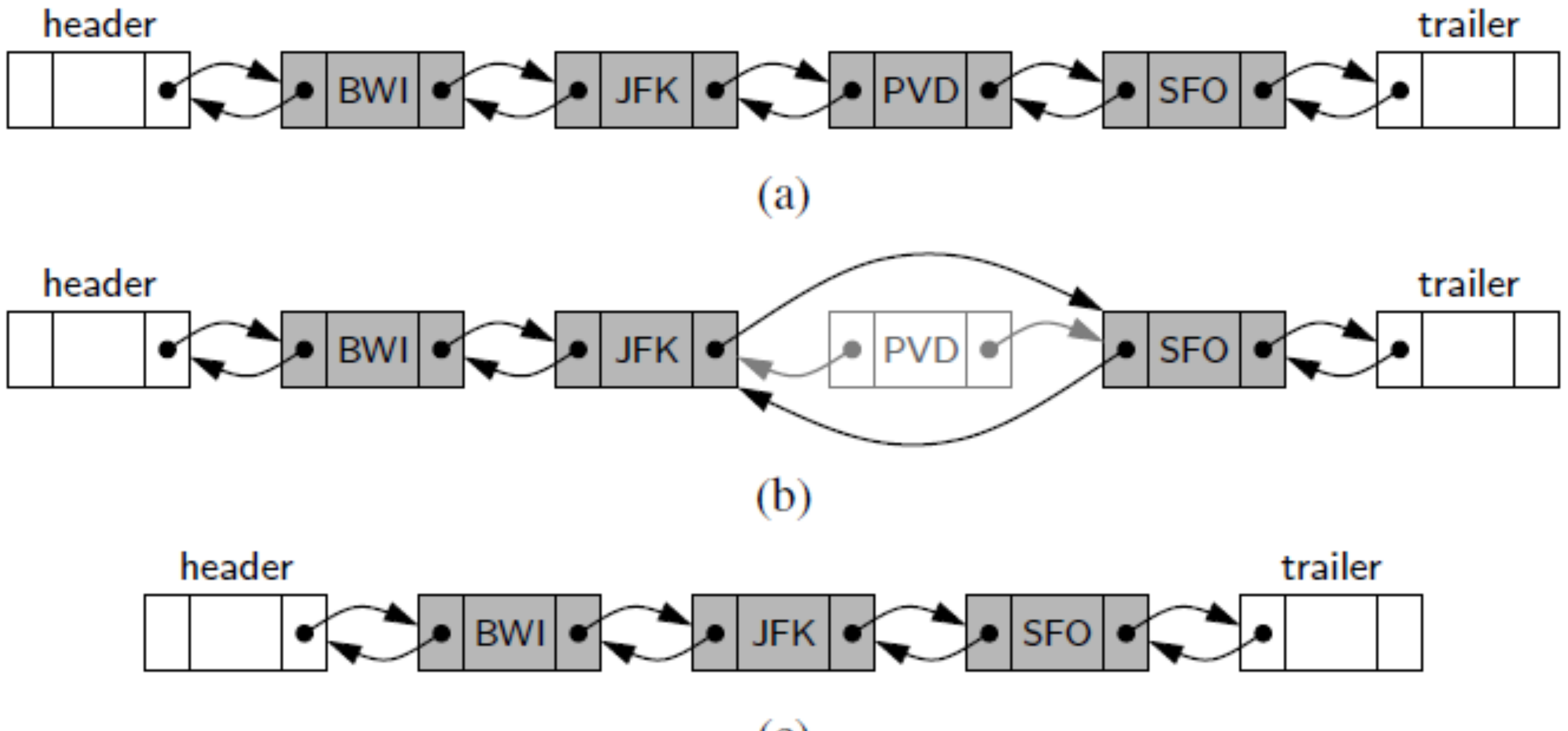


(b)



(c)

Deleting from the doubly linked list



(c)

Code for the doubly linked list

```
class Node:
    def __init__(self,element,prev,nxt):
        self.element = element
        self.prev = prev
        self.nxt = nxt

class DLList:

    def __init__(self):
        self.header = Node(None,None,None)
        self.trailer = Node(None,None,None)
        self.header.nxt = self.trailer
        self.trailer.prev = self.header
        self.size = 0

    def __len__(self):
        return self.size

    def is_empty(self):
        return self.size == 0
```

```python
def insert_between(self,e,predecessor,successor):
    newest = Node(e,predecessor,successor)
    predecessor.nxt = newest
    successor.prev = newest
    self.size+=1
    return newest

def delete_node(self,node):
    predecessor = node.prev
    successor = node.nxt
    predecessor.nxt = successor
    successor.prev = predecessor
    self.size -=1
    element = node.element
    node.prev = node.nxt = node.element = None
    return element

def iterate(self):
    pointer = self.header.nxt
    print('The elements in the list:')
    while pointer != self.trailer:
        print(pointer.element)
        pointer = pointer.nxt
```

```python
def main():
    d=DLList()
    d.__len__()

    newNode = d.insert_between(10,d.header,d.trailer)
    newNode = d.insert_between(20,newNode,d.trailer)
    newNode = d.insert_between(30,newNode,d.trailer)
    d.iterate()
    d.delete_node(d.header.nxt.nxt)
    d.iterate()
```

Bubble sort

- Bubble sort is a simple sorting algorithm

- Its general procedure is:

1) Iterate over a list of numbers, compare every element i with the following element i+1, and swap them if i is larger
2) Iterate over the list again and repeat the procedure in step 1, but ignore the last element in the list
3) Continuously iterate over the list, but each time ignore one more element at the tail of the list, until there is only one element left

Practice: Bubble sort over a standard list

```
def bubble(bubbleList):
    listLength = len(bubbleList)
    while listLength > 0:
        for i in range(listLength - 1):
            if bubbleList[i] > bubbleList[i+1]:
                buf = bubbleList[i]
                bubbleList[i] = bubbleList[i+1]
                bubbleList[i+1] = buf
        listLength -= 1
    return bubbleList


def main():
    bubbleList = [3, 4, 1, 2, 5, 8, 0, 100, 17]
    print(bubble(bubbleList))
```

Practice: Bubble sort over a singly linked list

Solution:

```python
from LinkedQueue import LinkedQueue

def LinkedBubble(q):
    listLength = q.size

    while listLength > 0:
        index = 0
        pointer = q.head
        while index < listLength-1:
            if pointer.element > pointer.pointer.element:
                buf = pointer.element
                pointer.element = pointer.pointer.element
                pointer.pointer.element = buf
            index += 1
            pointer = pointer.pointer
        listLength -= 1
    return q
```

```python
def outputQ(q):
    pointer = q.head

    while pointer:
        print(pointer.element)
        pointer = pointer.pointer

def main():
    oldList = [9,8,6,10,45,67,21,1]
    q = LinkedQueue()

    for i in oldList:
        q.enqueue(i)

    print('Before the sorting...')
    outputQ(q)

    q = LinkedBubble(q)
    print()
    print('After the sorting...')
    outputQ(q)
```

Quick sort

- Quick sort is a widely used algorithm, which is more efficient than bubble sort

- The main procedure of quick sort algorithm is:

1) Pick an element, called a pivot, from the array
2) Partitioning: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation
3) Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values

Practice: Quick sort over a standard list

```
def quickSort(L, low, high):
    i = low
    j = high
    if i >= j:
        return L
    key = L[i]
    while i < j:
        while i < j and L[j] >= key:
            j = j-1
        L[i] = L[j]
        while i < j and L[i] <= key:
            i = i+1
        L[j] = L[i]
    L[i] = key
    quickSort(L, low, i-1)
    quickSort(L, j+1, high)
    return L
```

Practice: Quick sort over a singly linked list