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

Tongxin Li
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

More about linked lists

Common Questions

Why do we need to build linked lists in Python?

When are linked lists preferred over lists?

Asked 2 years, 11 months ago

Modified 2 years, 11 months ago Viewed 1k times

4

I'm learning data structures and algorithms using Python. I've learnt that the advantage of linked list is that it does not have a maximum number of nodes, unlike arrays in other languages.

1. Since Python automatically resizes our lists for us, the advantage has been abstracted away for us.
2. Therefore, I've always thought that the only advantage linked lists have was that adding a node at the front or back of the linked list was O(1), whereas adding an element to a list could end up being O(n) from Python having to resize the array and copying every element over.

为什么要用python实现链表？

最近学校在学“python及其数据结构”，学完用python实现链表后感觉完全没有实用性啊，实属困惑，链表的功能用python自己的list不是都能实现吗？

收起 ▲

Python 数据结构 Python 入门

12 关注 · 0 评论 · 2.4 万浏览 关注问题

默认 最新 全部内容 8

尾翼

你们学校安排这门课，主要是想让你们学习数据结构，学习链表之类的数据结构的关键不在于用什么语言实现，而是让你们理解，你所谓那些list等已...

3 赞同 2018-10-15

夏韬

python作为 数据结构 这部分知识的载体 传授给你。 简化了你的阅读。 如果你想复杂点，有意义一点，请出门右转谭浩强C。

1 赞同 2018-10-16

Common Questions

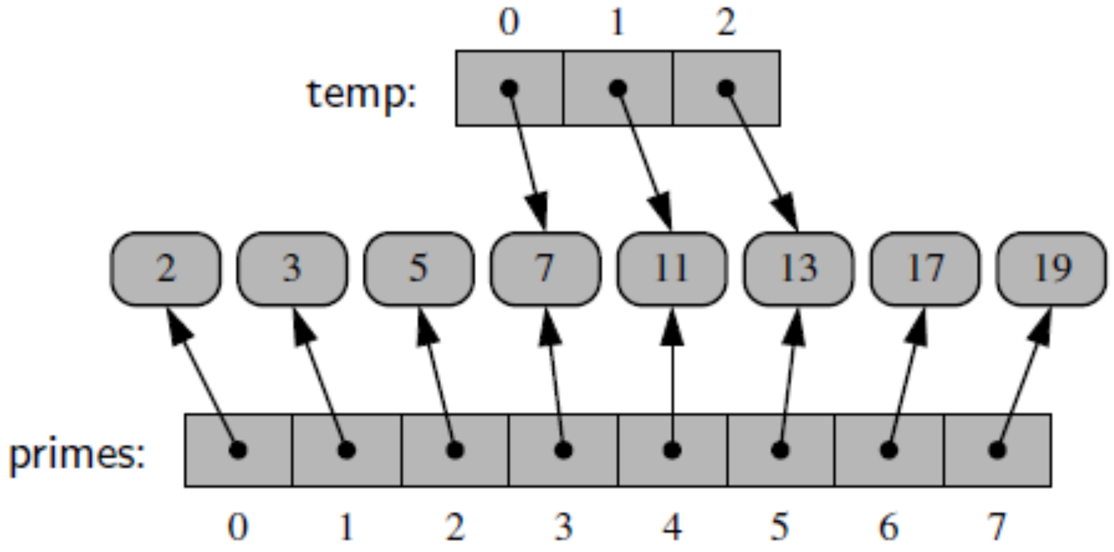
- Linked list is a **classic** data structure as a good example.
- Rarely used in **high** level programming languages like Python. As a practitioner, you may never use it in Python.
- We use Python to demonstrate the **concept**, so you know what it is when using non-Python languages.

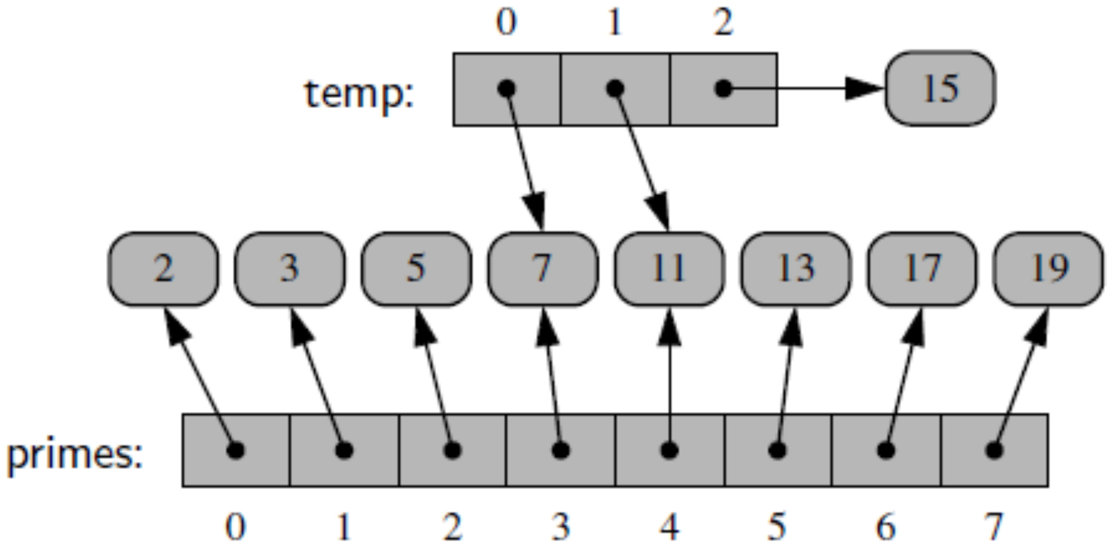
List in Python is a referential structure

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


1546964720
1546964752
1546964784
1546964816
1546964848
```

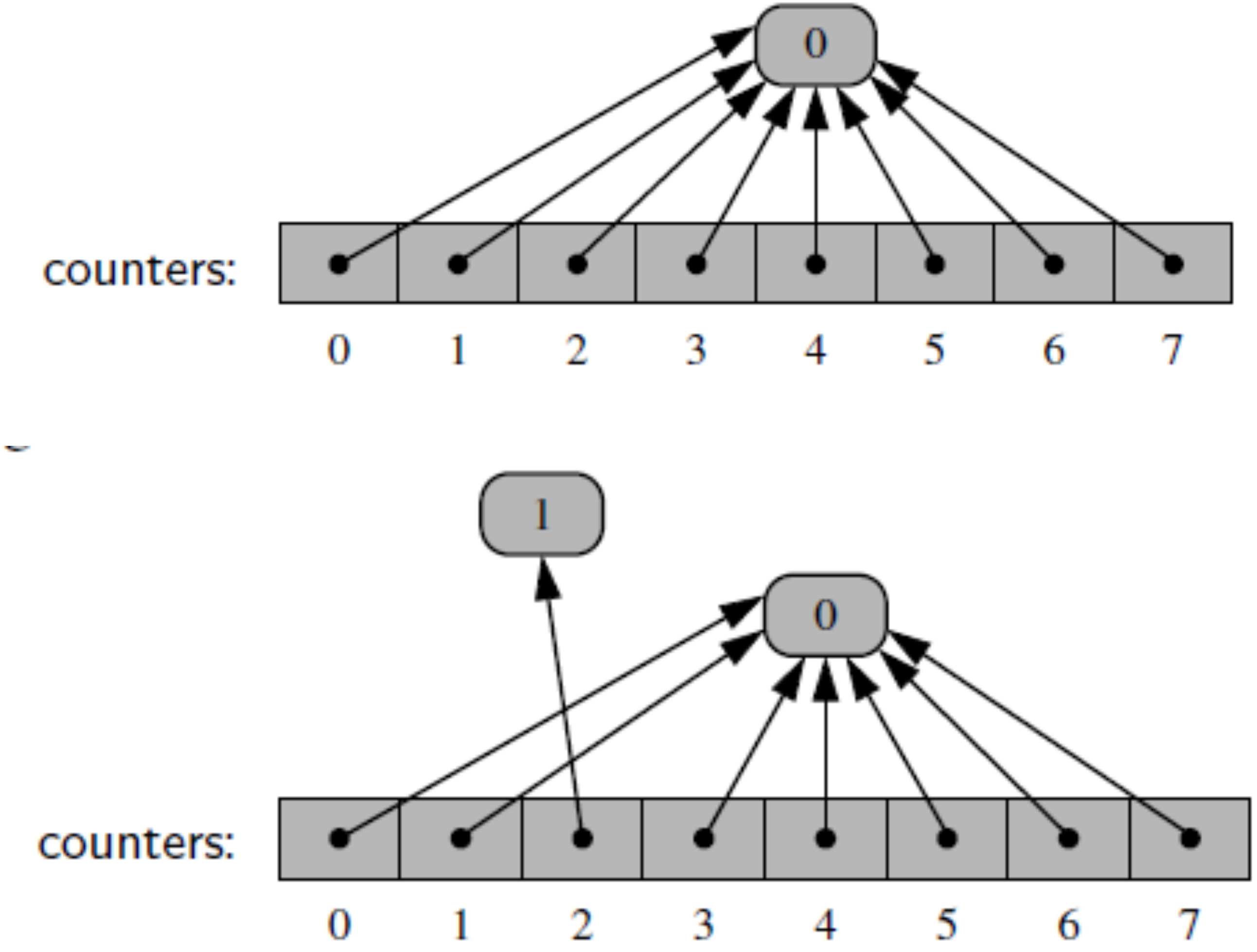
```
>>> a.insert(2,10)
>>> a
[1, 2, 10, 3, 4, 5]
>>> for i in range(0,6):
        print(id(a[i]))


1546964720
1546964752
1546965008
1546964784
1546964816
1546964848
```

List in Python is a referential structure





List in Python is a referential structure



Compact array

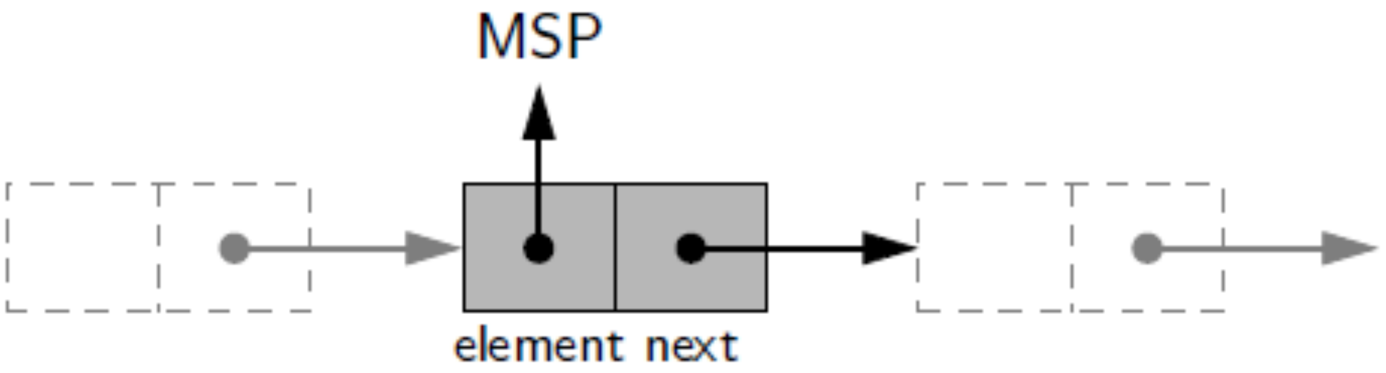
- A collection of numbers are usually stored as a compact array in languages such as C/C++ and Java

- 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)

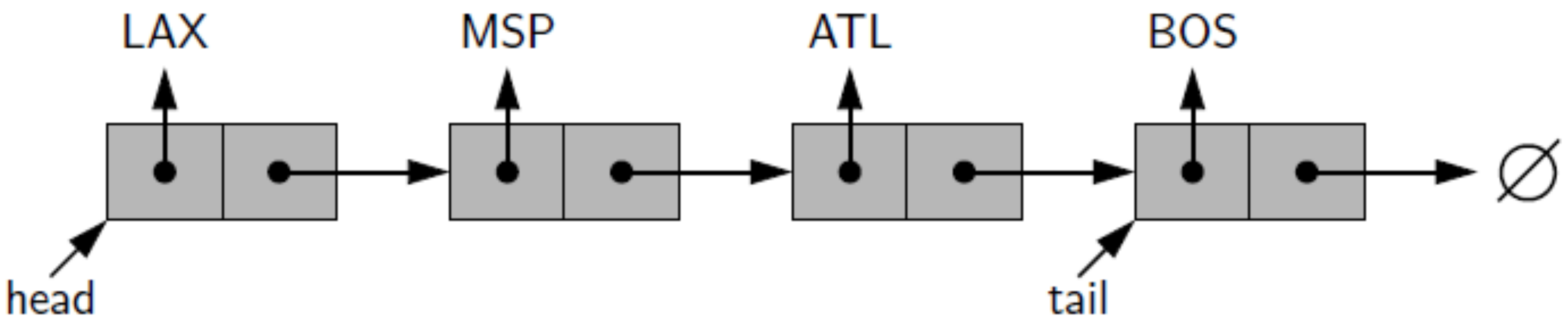
Linked List

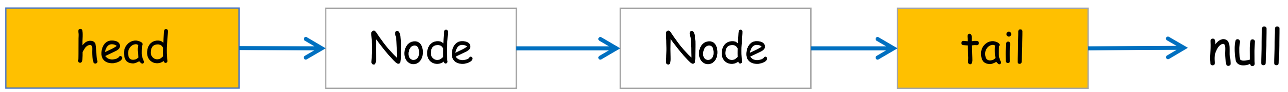
- A singly linked list, in its simplest form, is a collection of nodes that collectively form a linear sequence
- Each node stores a reference to an object that is an element of the sequence, as well as a reference to the next node of the list



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

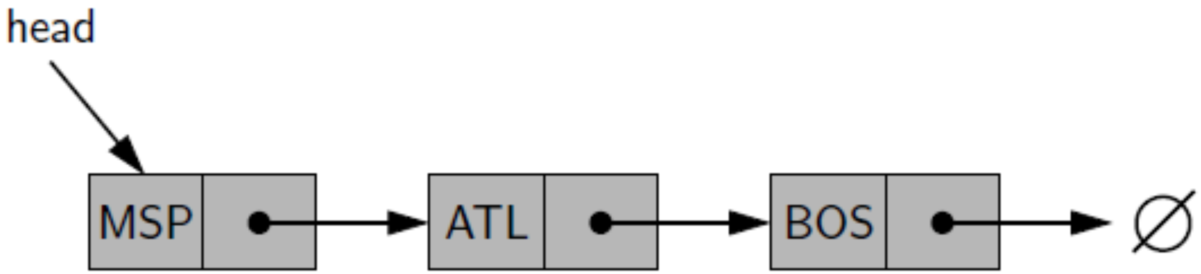




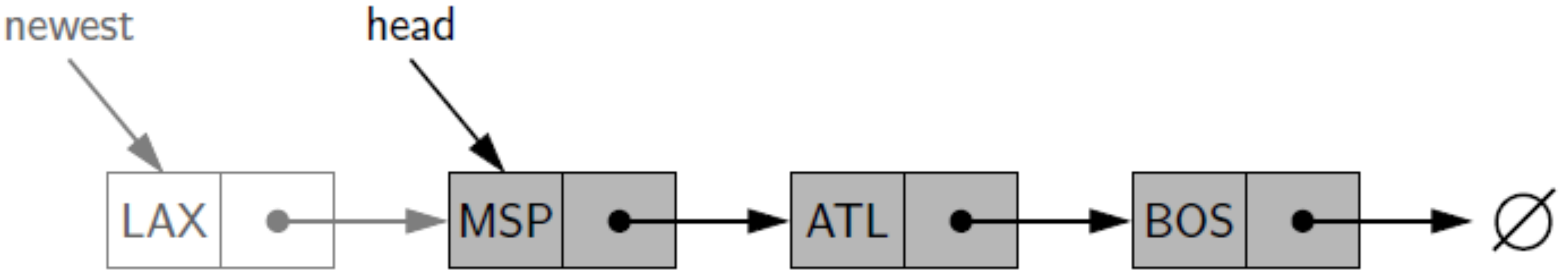

```python
class Node:
    def __init__(self,element,pointer=None):
        self.element = element
        self.pointer = pointer
```

```python
head = Node(0)
node1 = Node(1)
node2 = Node(2)
tail = Node(3)

head.pointer = node1
node1.pointer = node2
node2.pointer = tail

p = head
while(p!=None):
    print(p.element)
    p = p.pointer
```

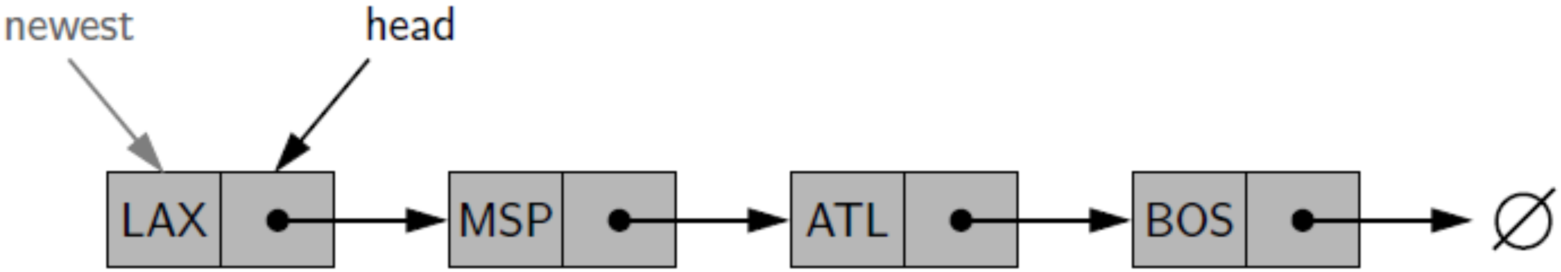
Inserting an Element at the Head of a Singly Linked List



(a)



(b)

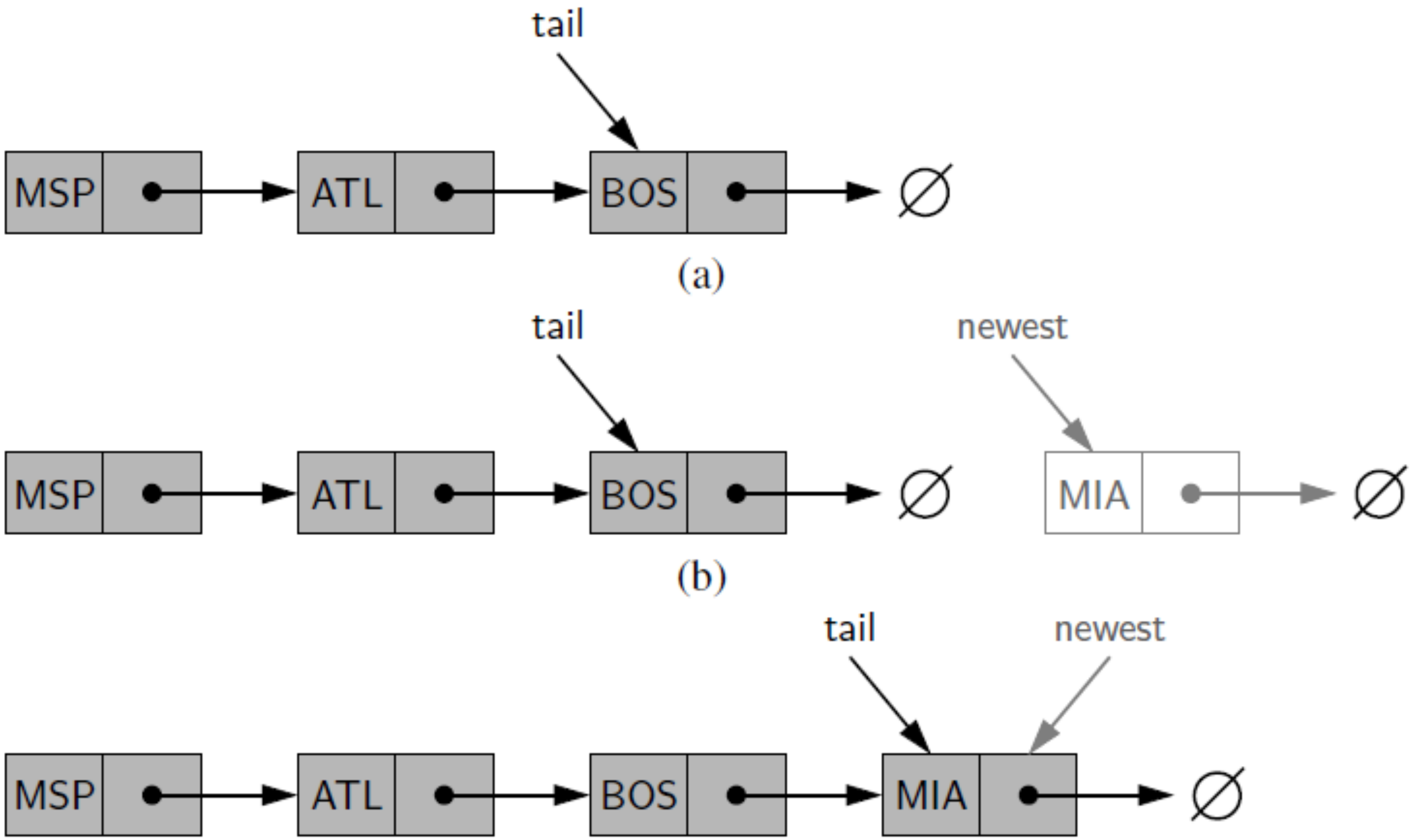


Pseudo code for inserting a node at the head

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

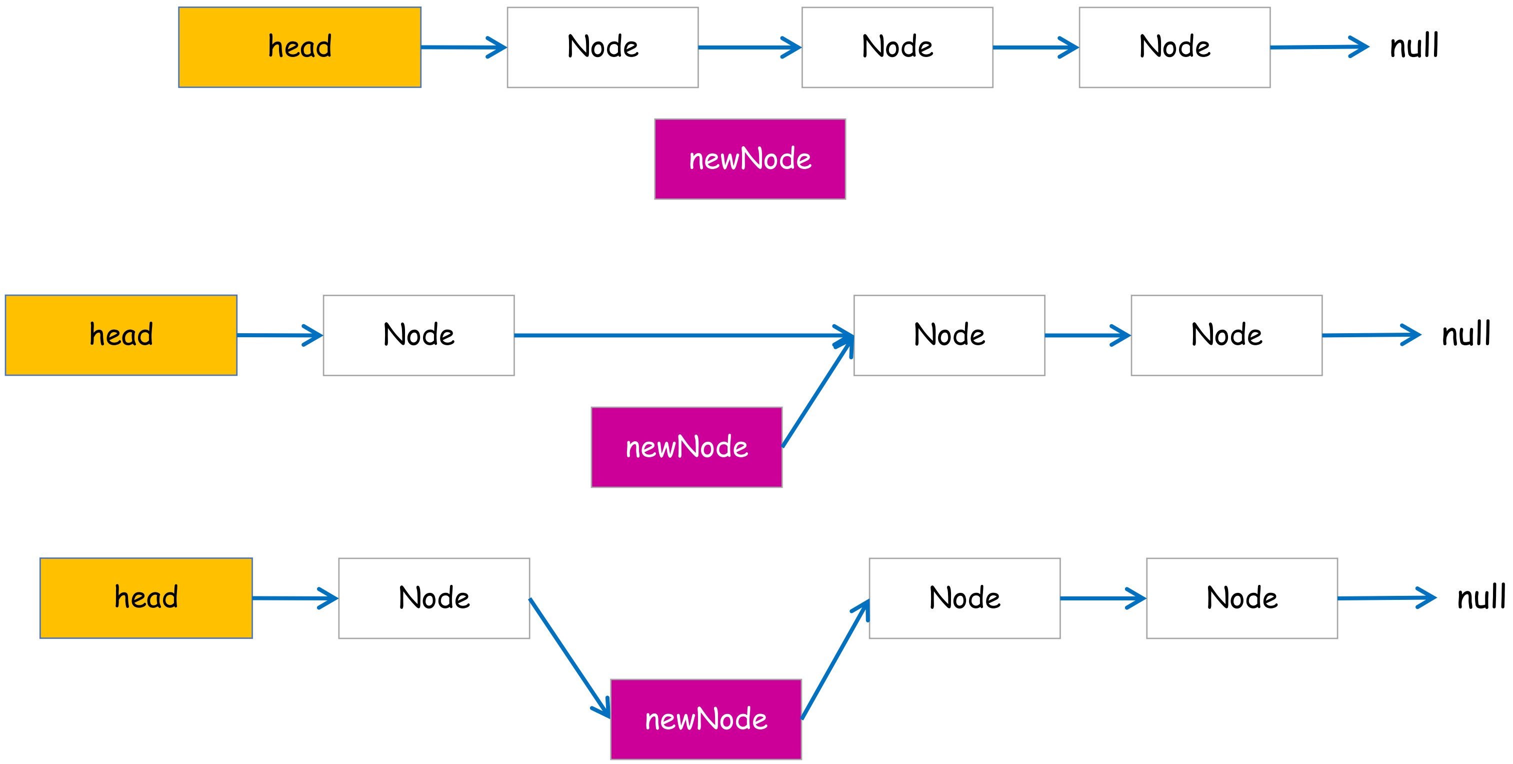


Pseudo code for inserting at the tail

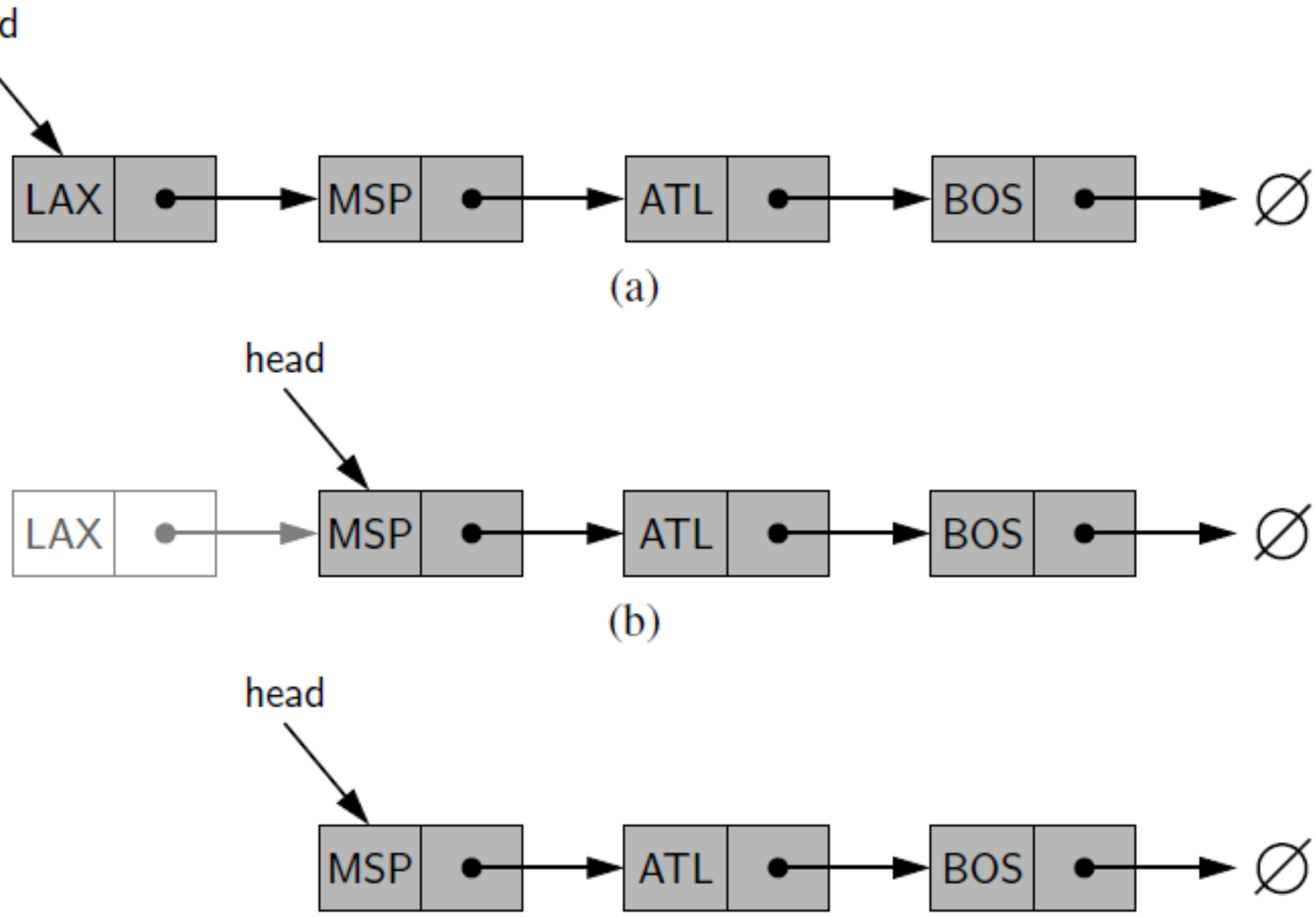
```
Algorithm add_last(L,e):
    newest = Node(e)  {create new node instance storing reference to element e}
    newest.next = None          {set new node's next to reference the None object}
    L.tail.next = newest                    {make old tail node point to new node}
    L.tail = newest      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

Linked list :: insert

- Add a new element to the list



Removing an Element from the head of a Singly Linked List

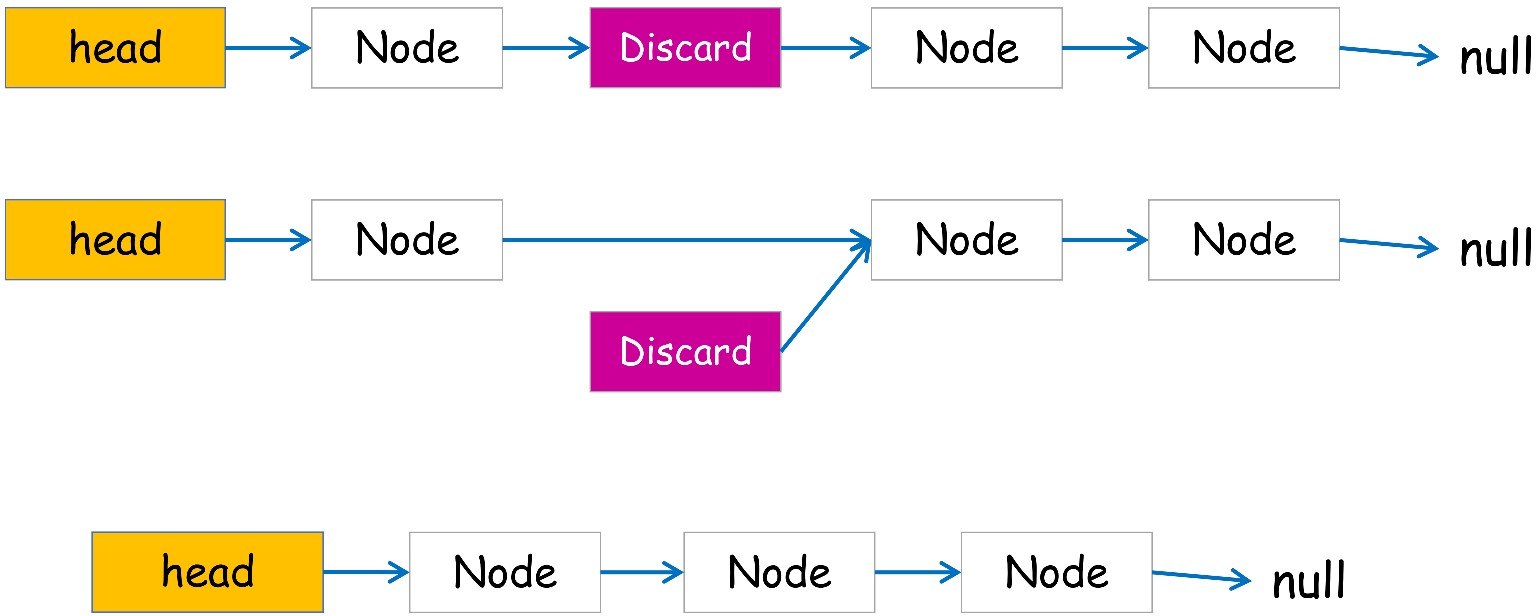


Pseudo code for removing a node from the head

```
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}
```

Linked list :: delete

- Delete a node from the list



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
```

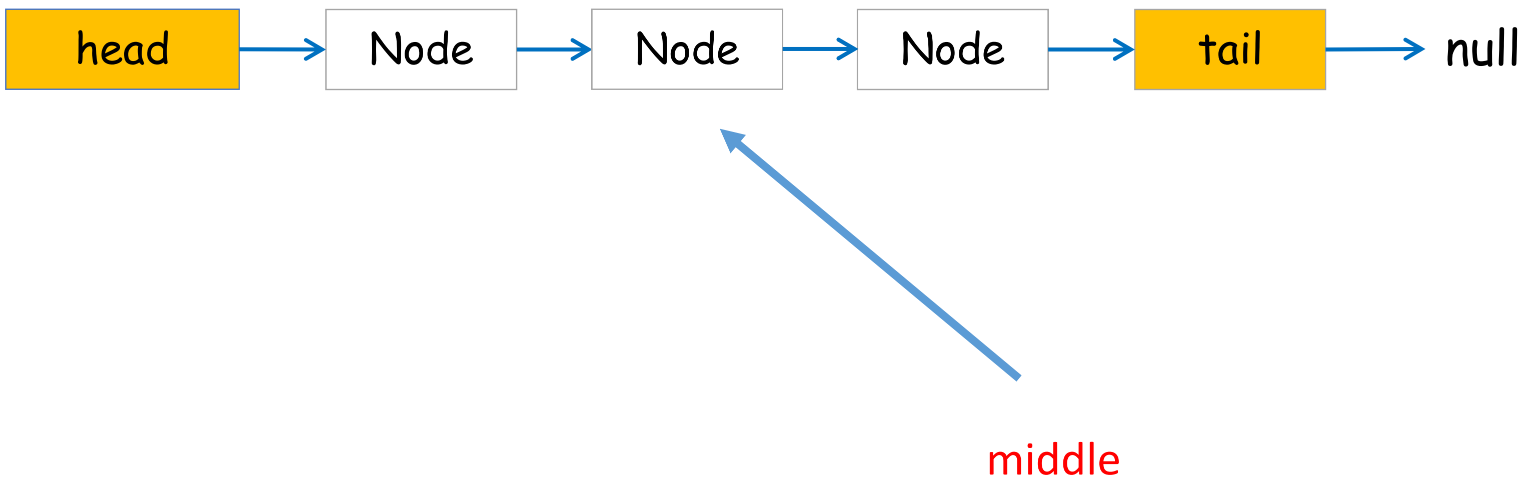
```
def top(self):
    if self.is_empty():
        print('Stack is empty.')
    else:
        return self.head.element

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
```

Practice: how to find the middle node (Optional)



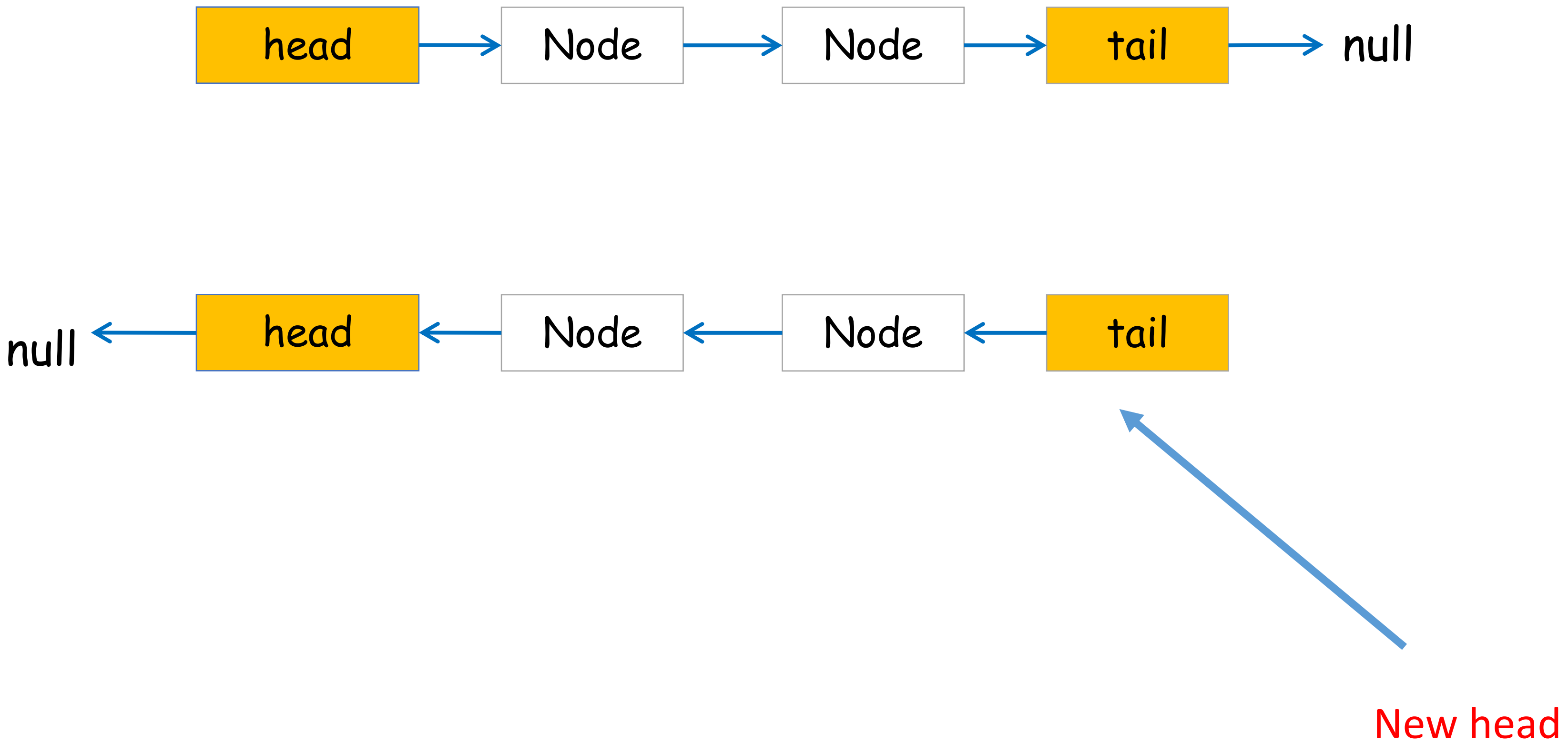
Practice: how to find the middle node (Optional)

```python
def findMiddleByLength(head):
    # First loop to calculate the length of the linked list
    length = 0
    current = head
    while current:
        length += 1
        current = current.next

    # Calculate the middle index for the first middle node
    middle_index = (length - 1) // 2

    # Second loop to traverse to the middle node
    current = head
    index = 0
    while index < middle_index:
        current = current.next
        index += 1

    print("The middle node's data is:", current.data)
    return current
```

Practice: how to find the middle node (Optional)

```python
# Define the Node class
class Node:
    def __init__(self, data):
        self.data = data  # Store data
        self.next = None  # Pointer to the next node

# Function to find the middle node of the linked list
def findMiddle(head):
    slow = head  # Slow pointer moves one step at a time
    fast = head  # Fast pointer moves two steps at a time
    while fast and fast.next:
        slow = slow.next          # Move slow pointer by one
        fast = fast.next.next     # Move fast pointer by two
    print("The middle node's data is:", slow.data)
    return slow
```

How to reverse a linked list (Optional)



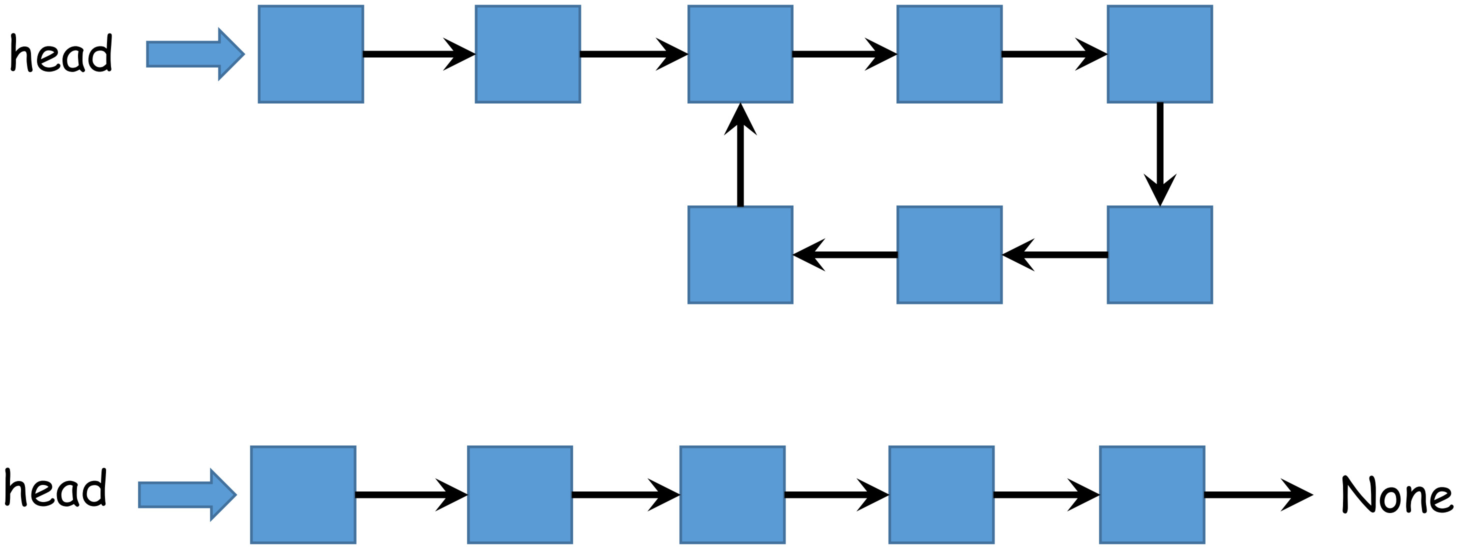
How to reverse a linked list (Optional)

```python
class Node:
    def __init__(self, data):
        self.data = data  # Store data
        self.next = None  # Pointer to the next node

# Function to reverse the linked list
def reverseLinkedList(head):
    prev = None
    current = head
    while current:
        next_node = current.next      # Save the next node
        current.next = prev           # Reverse the link
        prev = current                # Move prev to current
        current = next_node           # Move to next node
    return prev  # New head of the reversed list
```

How to reverse a linked list - recursion

```python
# Define the Node class
class Node:
    def __init__(self, data):
        self.data = data  # Store data
        self.next = None  # Pointer to the next node

# Function to reverse the linked list recursively
def reverseLinkedListRec(head):
    # Base case: if head is empty or only one node, it's already reversed
    if head is None or head.next is None:
        return head

    # Recursively reverse the rest of the list
    rest = reverseLinkedListRec(head.next)

    # Adjust the pointers
    head.next.next = head
    head.next = None

    # Return the new head of the reversed list
    return rest
```

Practice: check if there exists a cycle in a linked list (Optional)



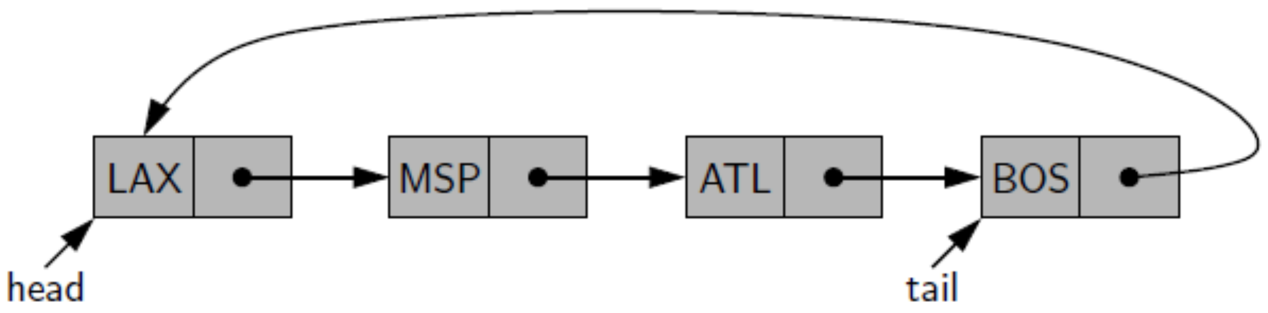
Practice: check if there exists a cycle in a linked list (Optional)

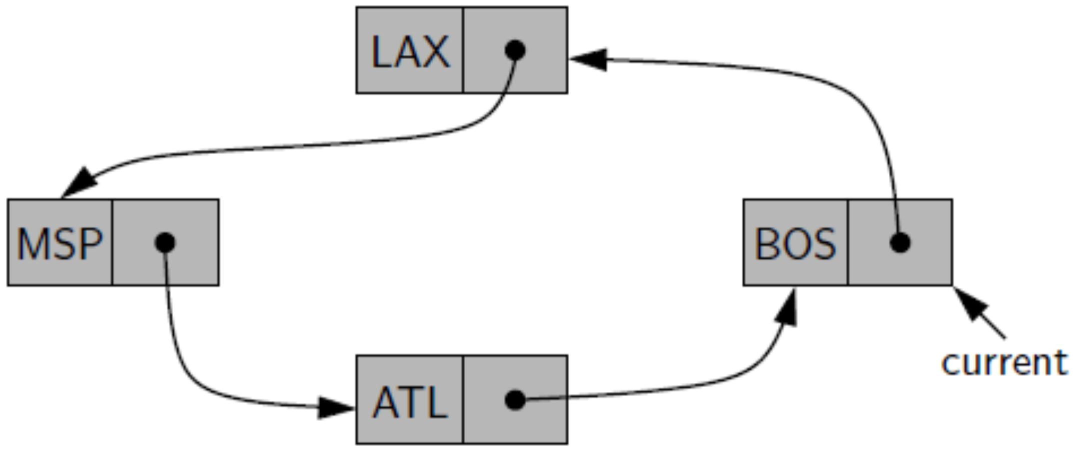
```python
# Define the Node class
class Node:
    def __init__(self, data):
        self.data = data  # Store data
        self.next = None  # Pointer to the next node

# Function to check if the linked list has a cycle
def hasCycle(head):
    slow = head  # Tortoise moves one step at a time
    fast = head  # Hare moves two steps at a time
    while fast and fast.next:
        slow = slow.next           # Move slow pointer by one
        fast = fast.next.next      # Move fast pointer by two
        if slow == fast:
            print("Cycle detected at node with data:", slow.data)
            return True
    print("No cycle detected.")
    return False
```

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

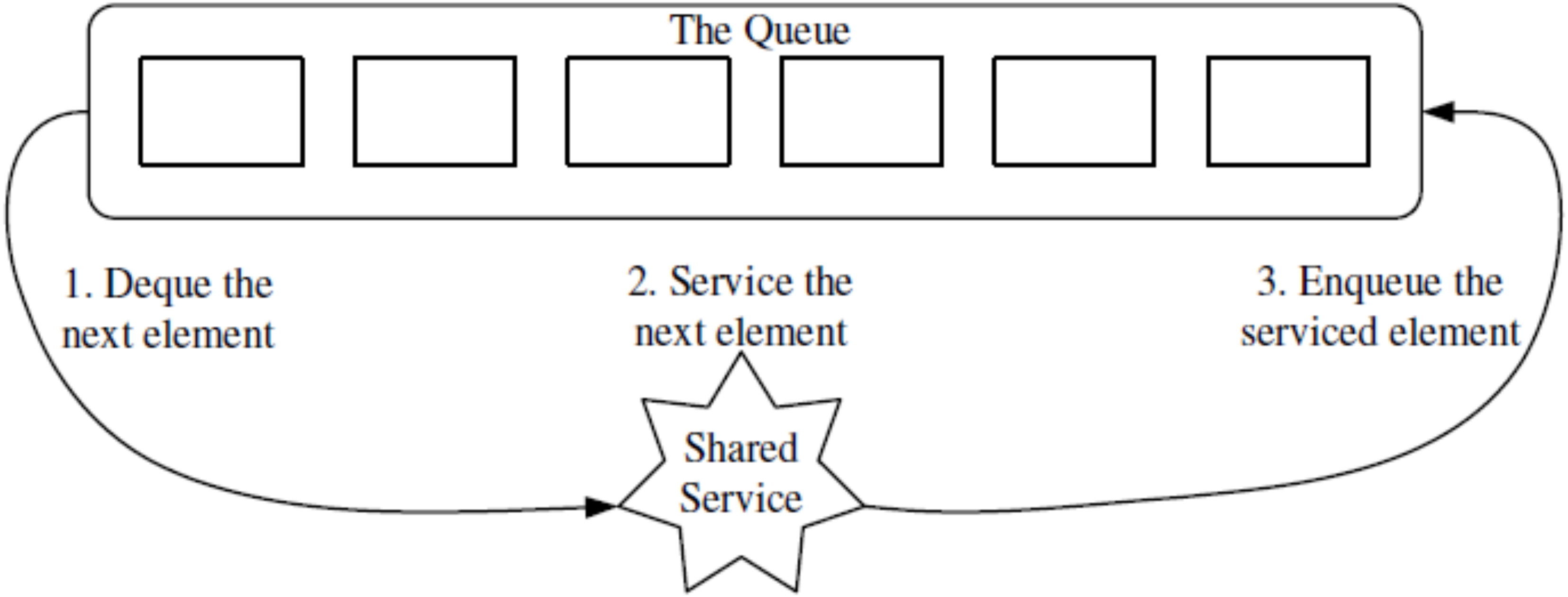
- A round-robin scheduler iterates through a collection of elements in a circular fashion and “serves” each element by performing a given action on it

- 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     # Skip the old head
        self.__size -= 1
        return oldhead.element

def enqueue(self, e):
    newest = Node(e, None)
    if self.is_empty():
        newest.pointer = newest                       # A single self-pointed node
    else:
        newest.pointer = self.__tail.pointer
        self.__tail.pointer = newest                  # Insert after the tail!
    self.__tail = newest
    self.__size += 1
```

Skip the old head

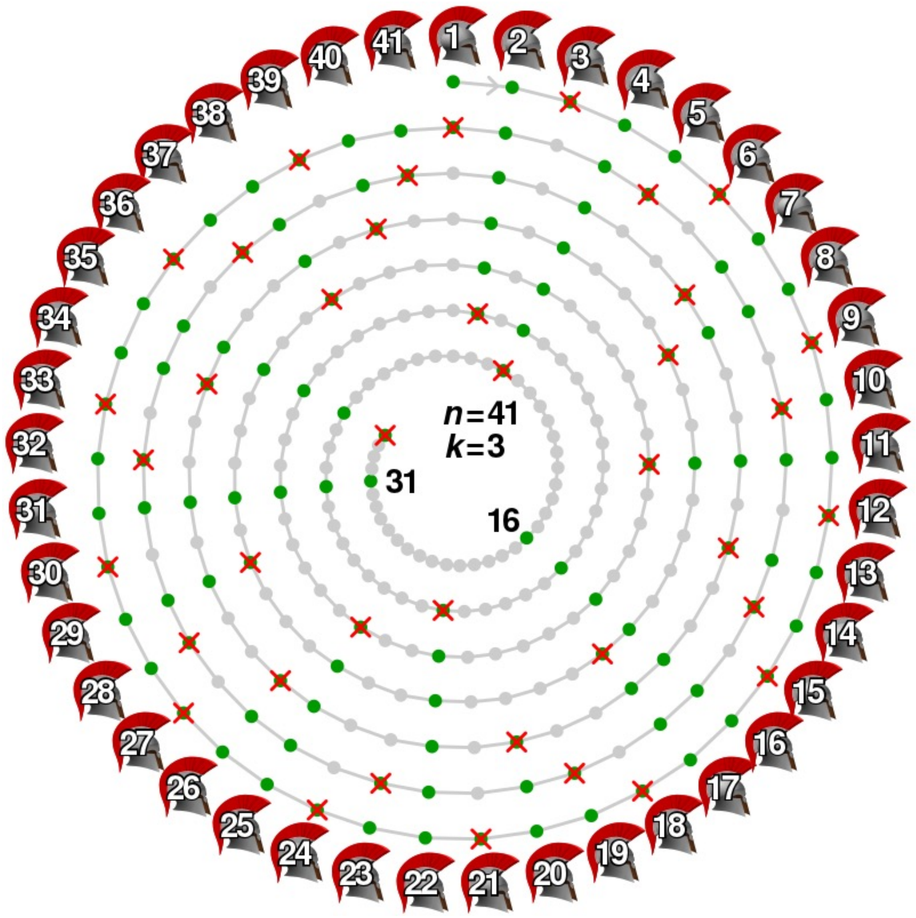
A single self-pointed node

Insert after the tail!

Exercise: The Josephus Problem

- The Josephus Problem
  - There are n people standing in a circle waiting to be executed. After the first man is executed, k - 1 people are skipped and the k-th man is executed. Then again, k-1 people are skipped and the k-th man is executed. The elimination proceeds around the circle (which is becoming smaller and smaller as the executed people are removed), until only the last man remains, who is given freedom.

  - The task is to choose the place in the initial circle so that you survive, given n and k.


1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
n=41
k=3
31
16

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

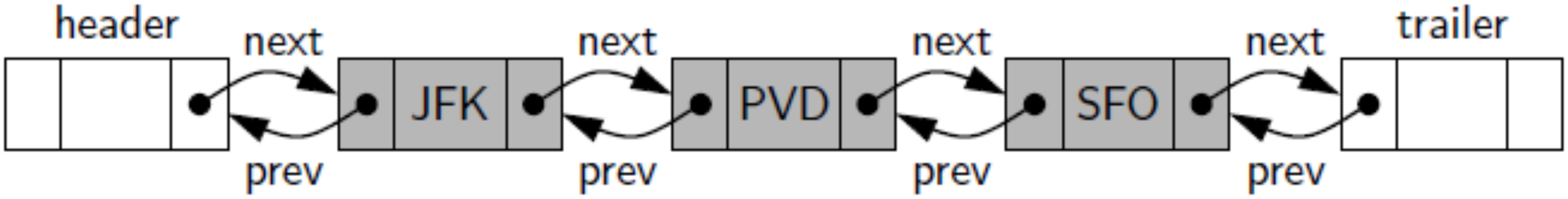
- But we cannot efficiently delete a node at the tail of the list

- 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

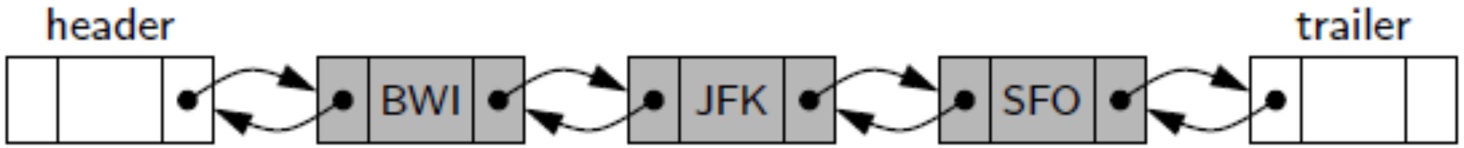
- This kind of data structure is called doubly linked list

Head and tail sentinels

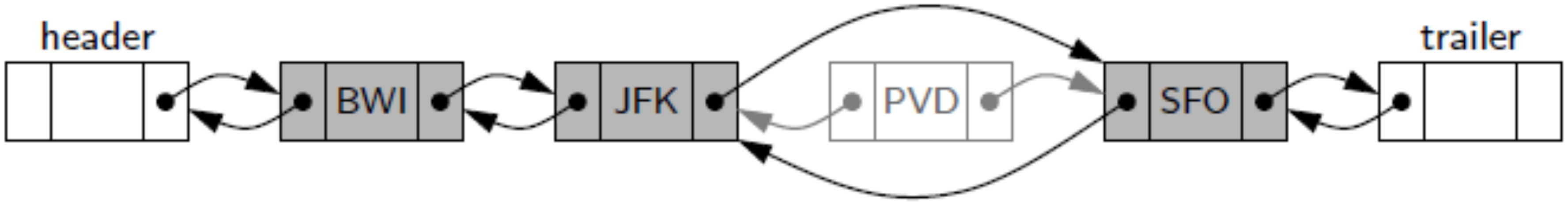
- In order to avoid some special cases when operating near the boundaries of a doubly linked list, it helps to add special nodes at both ends of the list: a header node at the beginning of the list, and a trailer node at the end of the list
- These "dummy" nodes are known as sentinels (or guards), and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



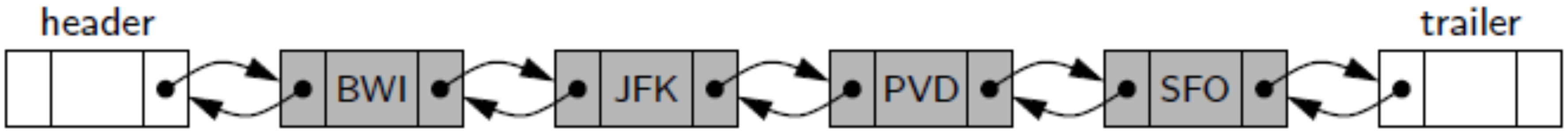
Inserting in the middle of a doubly linked list



(a)

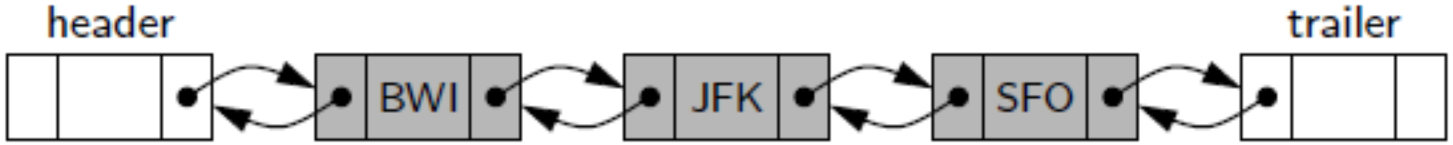


(b)

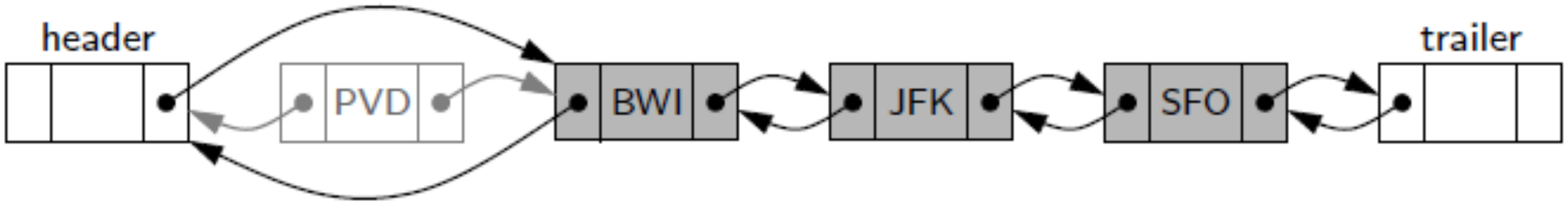


(c)

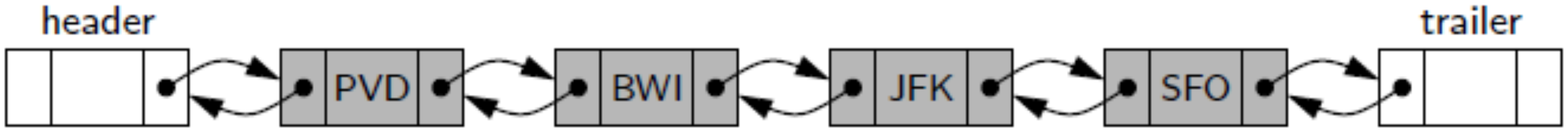
Inserting at the head of the doubly linked list



(a)

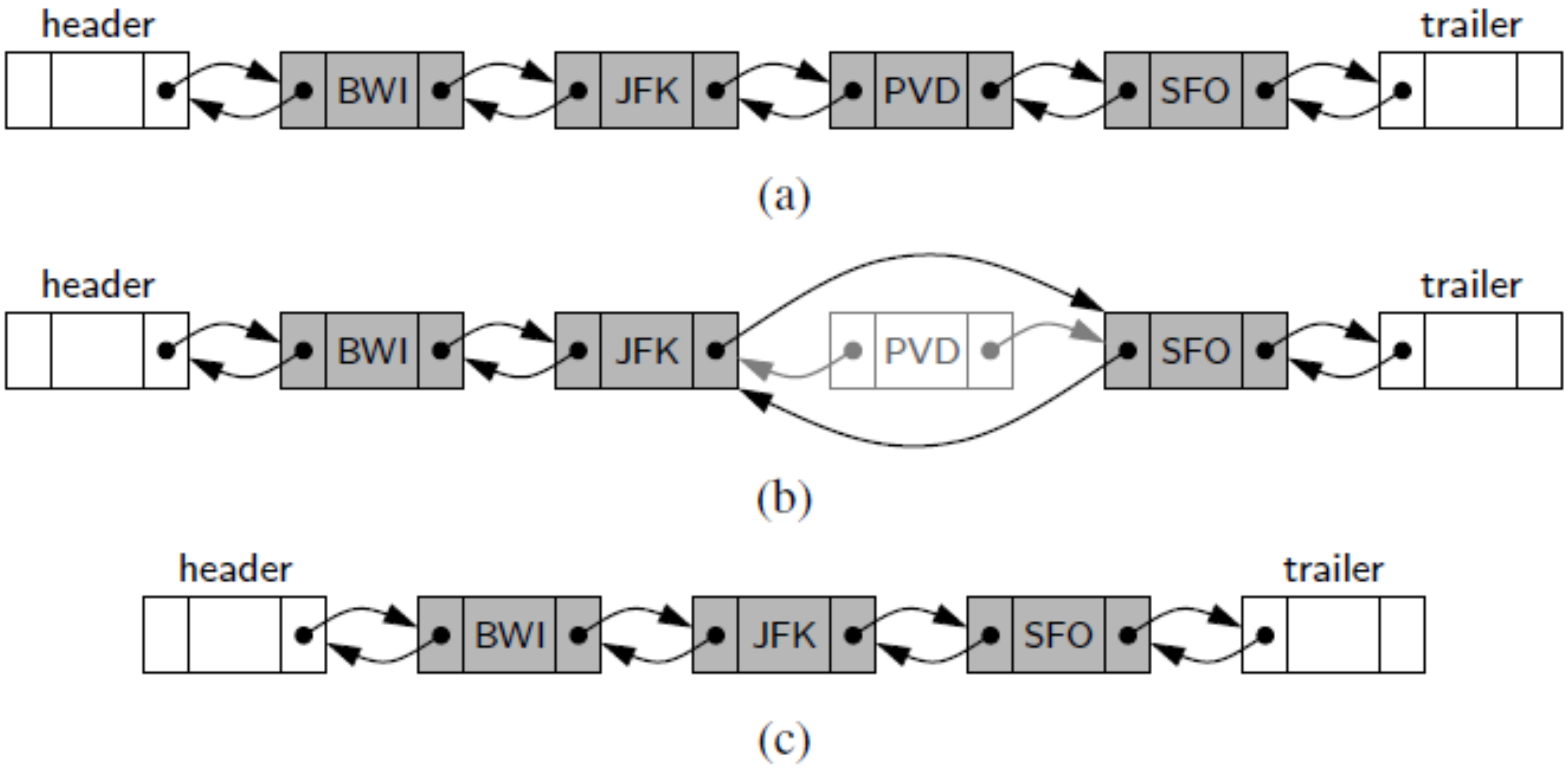


(b)



(c)

Deleting from the doubly linked list



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

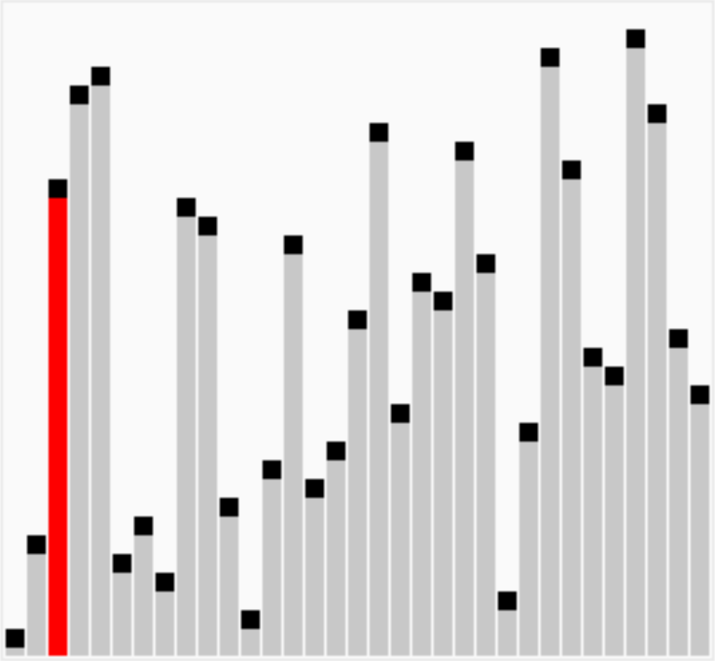
- Bubble sort is a simple sorting algorithm

- 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


3
4
2
1
9
8
7
6
5

A longer example



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
```

Swap values

```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

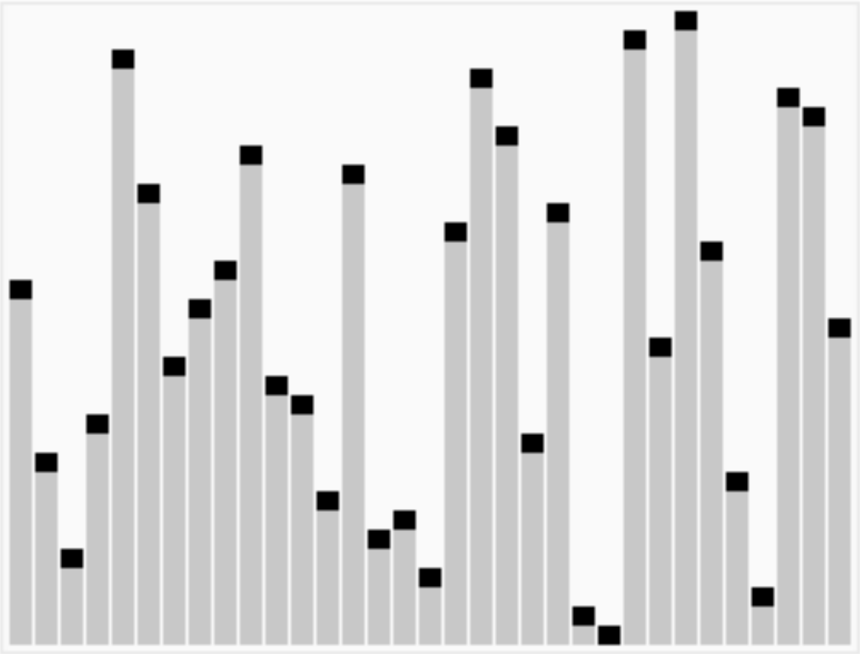
- Quick sort is a widely used algorithm, which is more efficient than bubble sort

- 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

Unsorted Array

A longer example



Practice: Quick sort over a standard list

```python
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

?