INTRODUCTION TO COMPUTER SCIENCE: PROGRAMMING METHODOLOGY

Lecture 11 Linked List and Sorting Algorithms

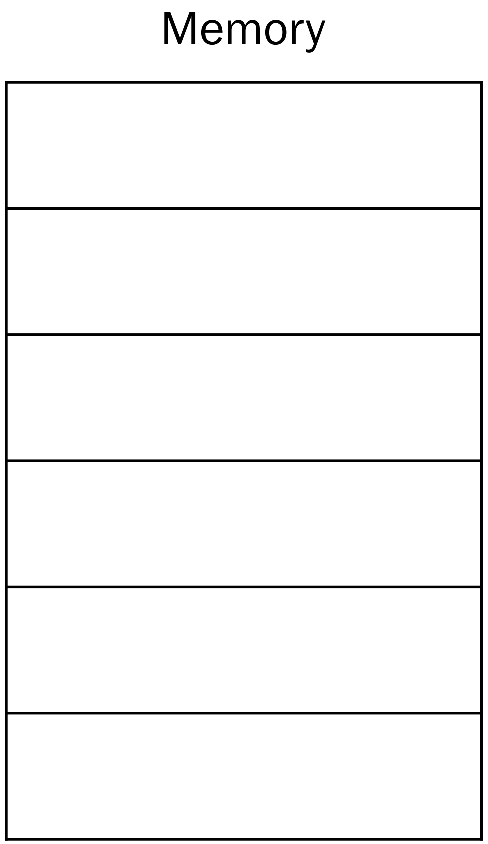
Prof. Wei Cai

School of Science and Engineering

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

WHY WE NEED ANOTHER LIST DATA TYPE?

- What is wrong with List in Python?
- Compact Array
  - Storing the bits that represent the primary data
  - Used in languages such as C/C++ and Java
  - The overall memory usage is very efficient

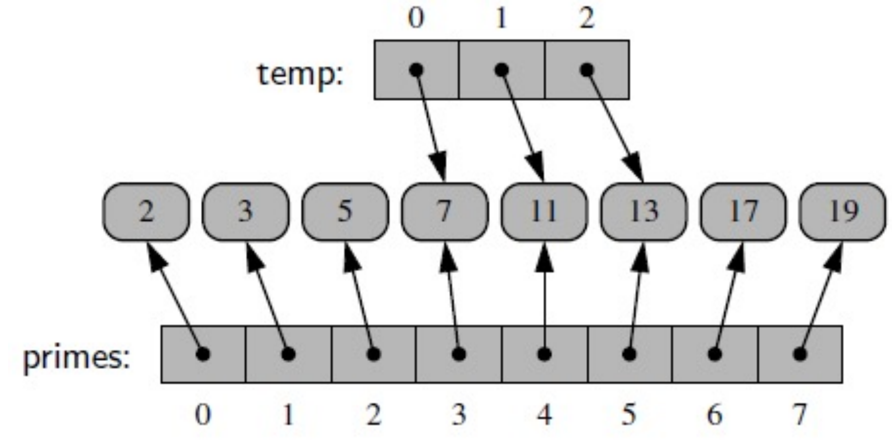


LIST IN PYTHON IS A REFERENTIAL STRUCTURE

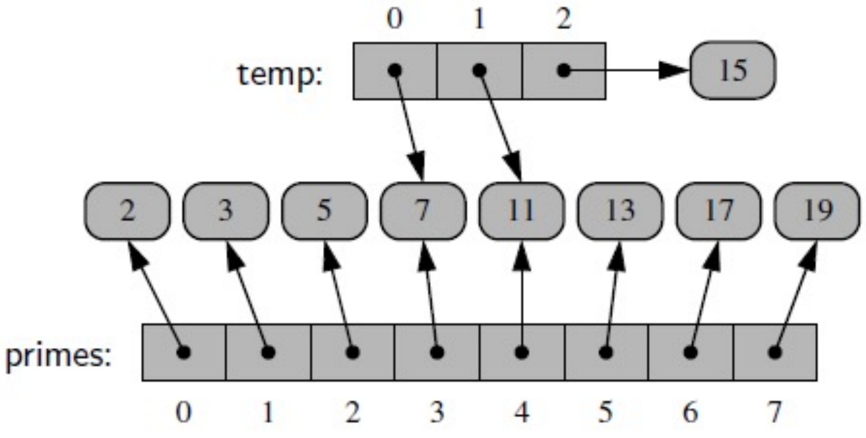
```
>>> a=[1, 2, 3, 4, 5]
>>> for i in range(0, 5):
        print(id(a[i]))


1546964720
1546964752
1546964784
1546964816
1546964848
```

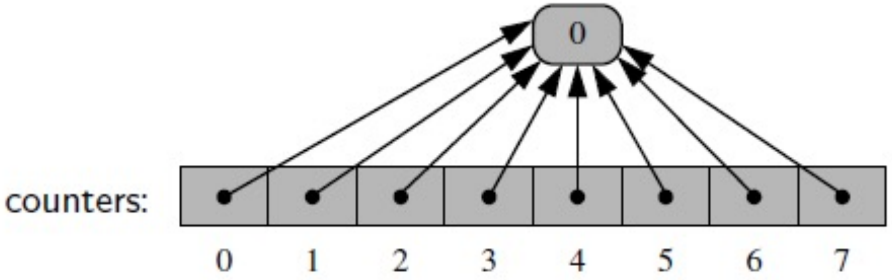
```
>>> a.insert(2, 10)
>>> a
[1, 2, 10, 3, 4, 5]
>>> for i in range(0, 6):
        print(id(a[i]))


1546964720
1546964752
1546965008
1546964784
1546964816
1546964848
```

LIST IN PYTHON IS A REFERENTIAL STRUCTURE

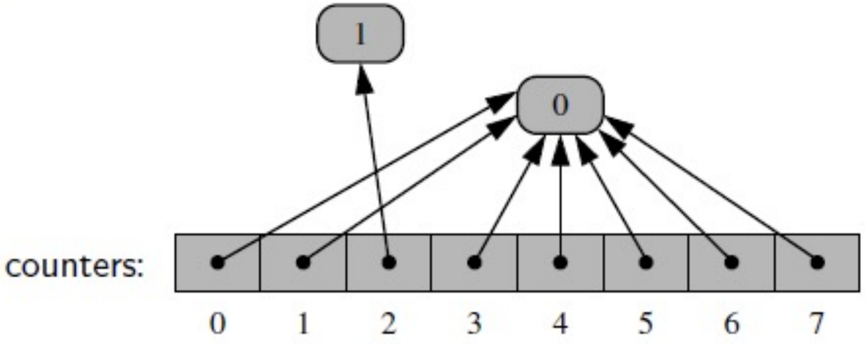


temp[2] = 15





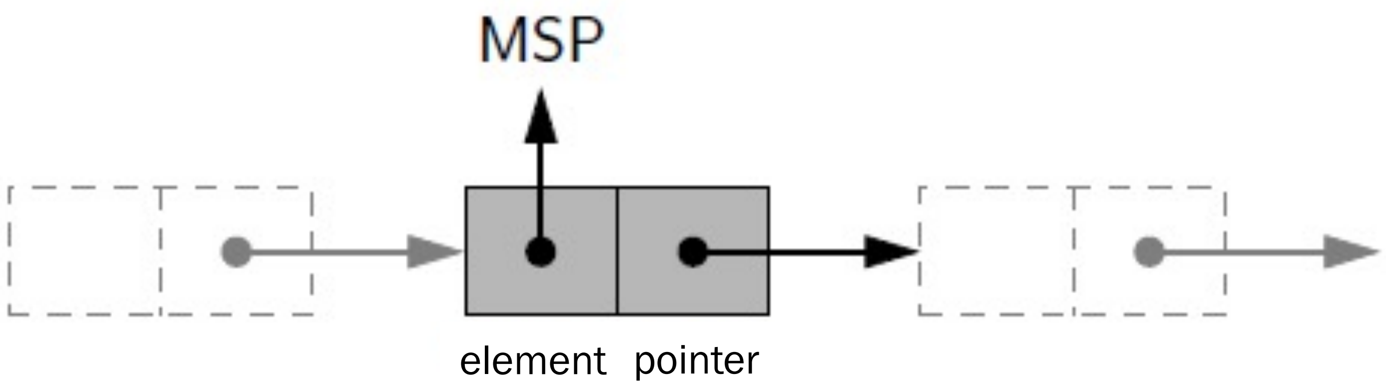
counters[2]=1



LIST IN PYTHON IS A REFERENTIAL STRUCTURE

- Python's list class is highly optimized, and often a great choice for storage
- The overall memory usage of Python list will be much higher because there is overhead devoted to the explicit storage of the sequence of memory references (in addition to the primary data)
- However, many programming languages do not support this kind of optimized list data type

LINKED LIST: A NODE

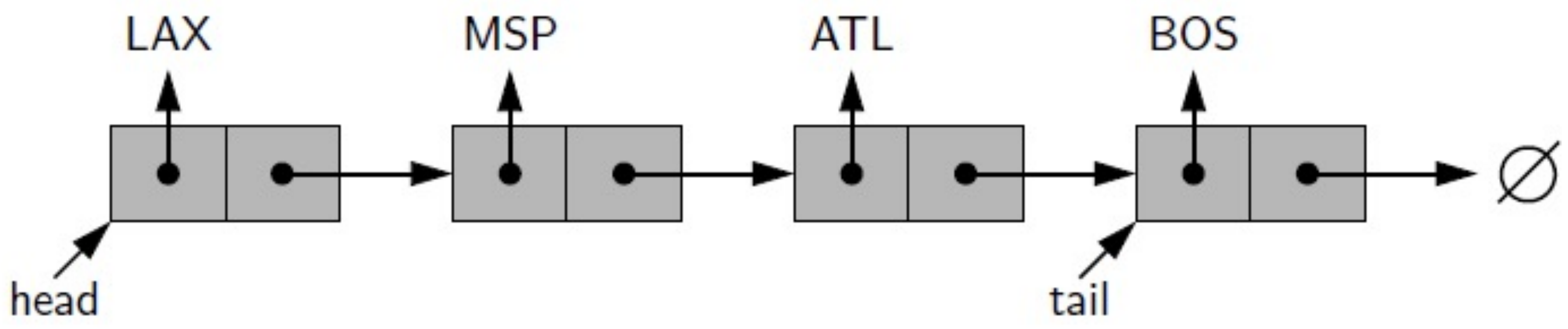


a reference to an object that is an element of the sequence

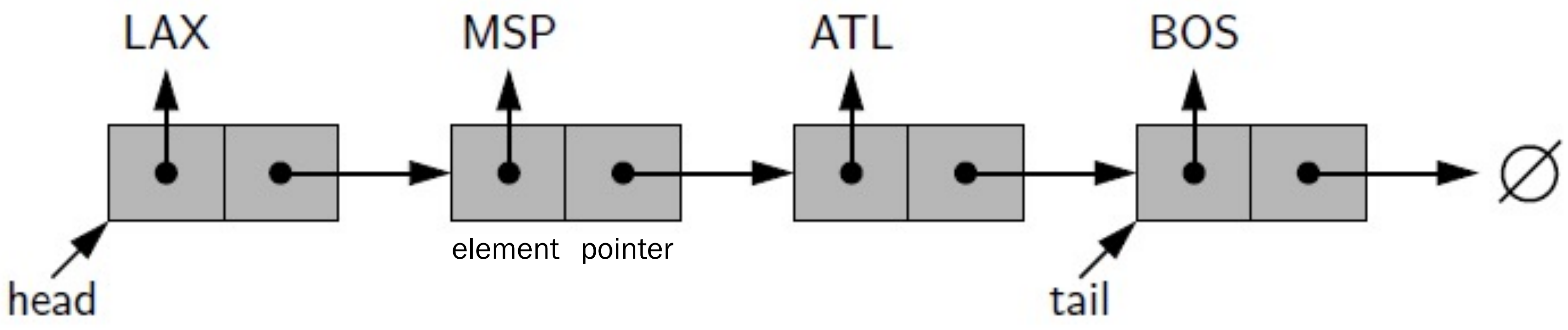
a reference to the next node of the list

SINGLY LINKED LIST

- a collection of nodes that collectively form a linear sequence



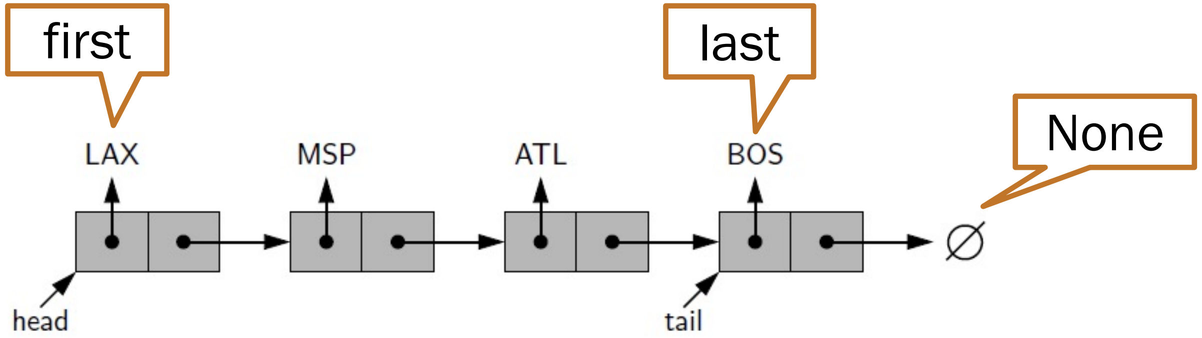
DEFINING A LINKED LIST IN PYTHON




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

class LinkedList:

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

```python
myList = LinkedList()

node1 = Node("BOS", None)
node2 = Node("ATL", node1)
node3 = Node("MSP", node2)
node4 = Node("LAX", node3)

myList.head = node4
myList.tail = node1
myList.size = 4
```

TRAVERSING THE LINKED LIST



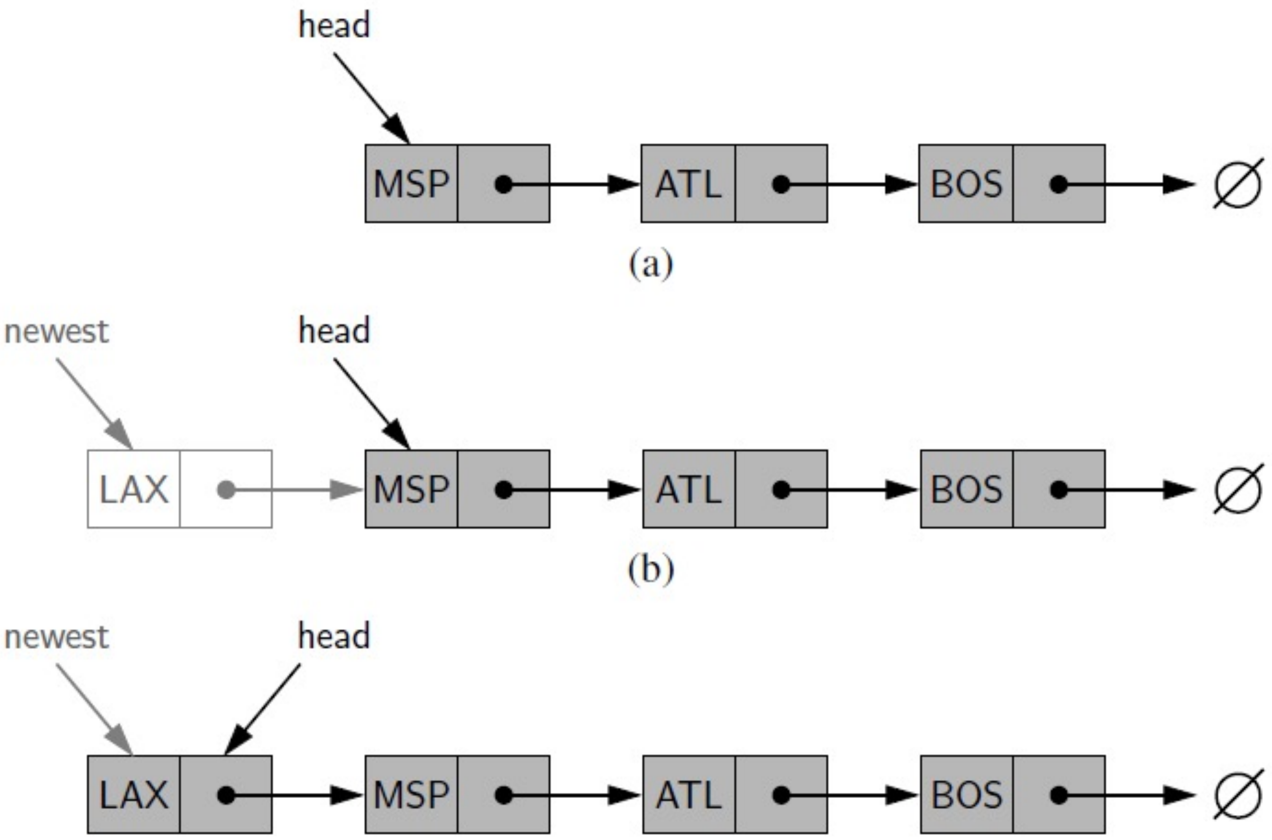
- By starting at the head, and moving from one node to another by following each node's *pointer* reference, we can reach the tail of the list
- also known as link hopping or pointer hopping

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

class LinkedList:

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

```python
myList = LinkedList()

node1 = Node("BOS", None)
node2 = Node("ATL", node1)
node3 = Node("MSP", node2)
node4 = Node("LAX", node3)

myList.head = node4
myList.tail = node1
myList.size = 4
```

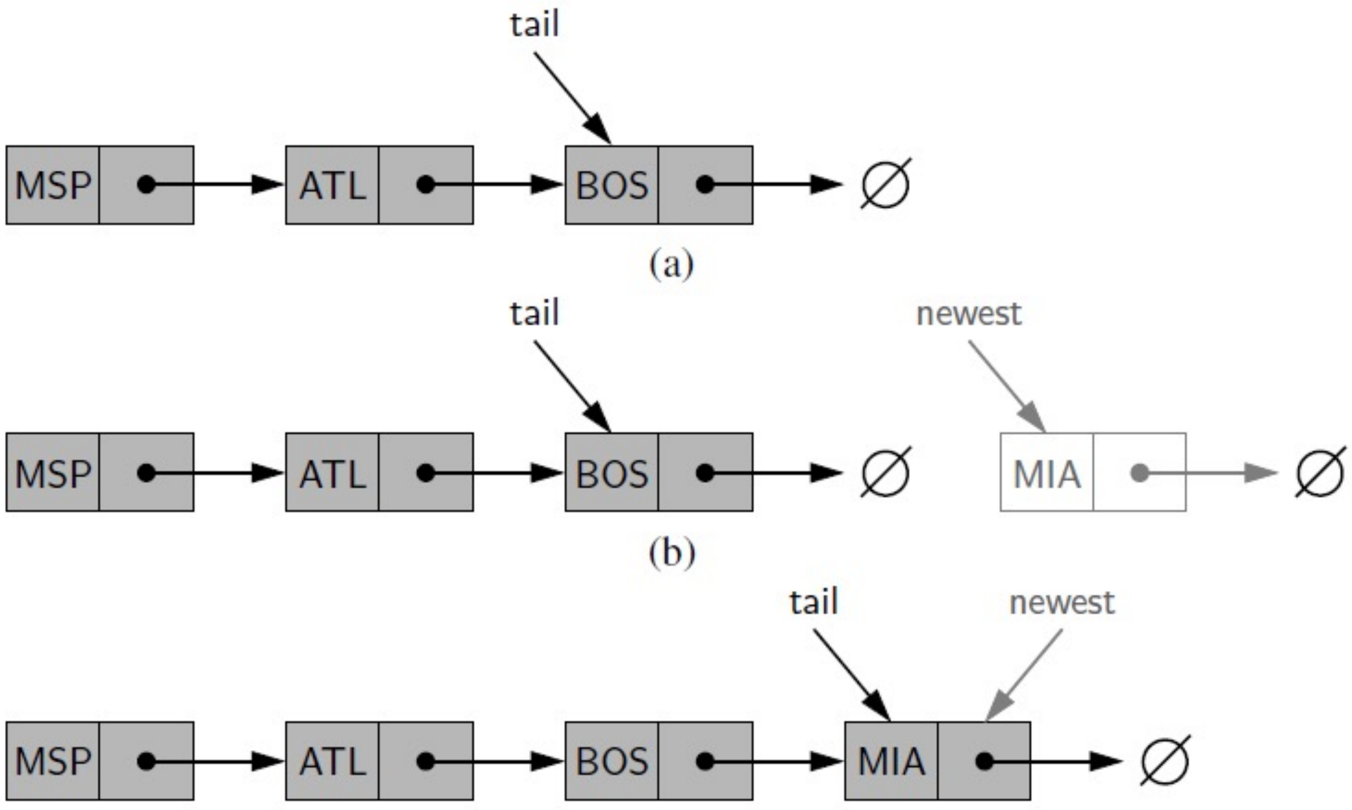
```python
i = myList.size
current = myList.head
while i>0:
    print(current.element)
    current = current.pointer
    i-=1
```

INSERTING AN ELEMENT AT THE HEAD OF A SINGLY LINKED LIST



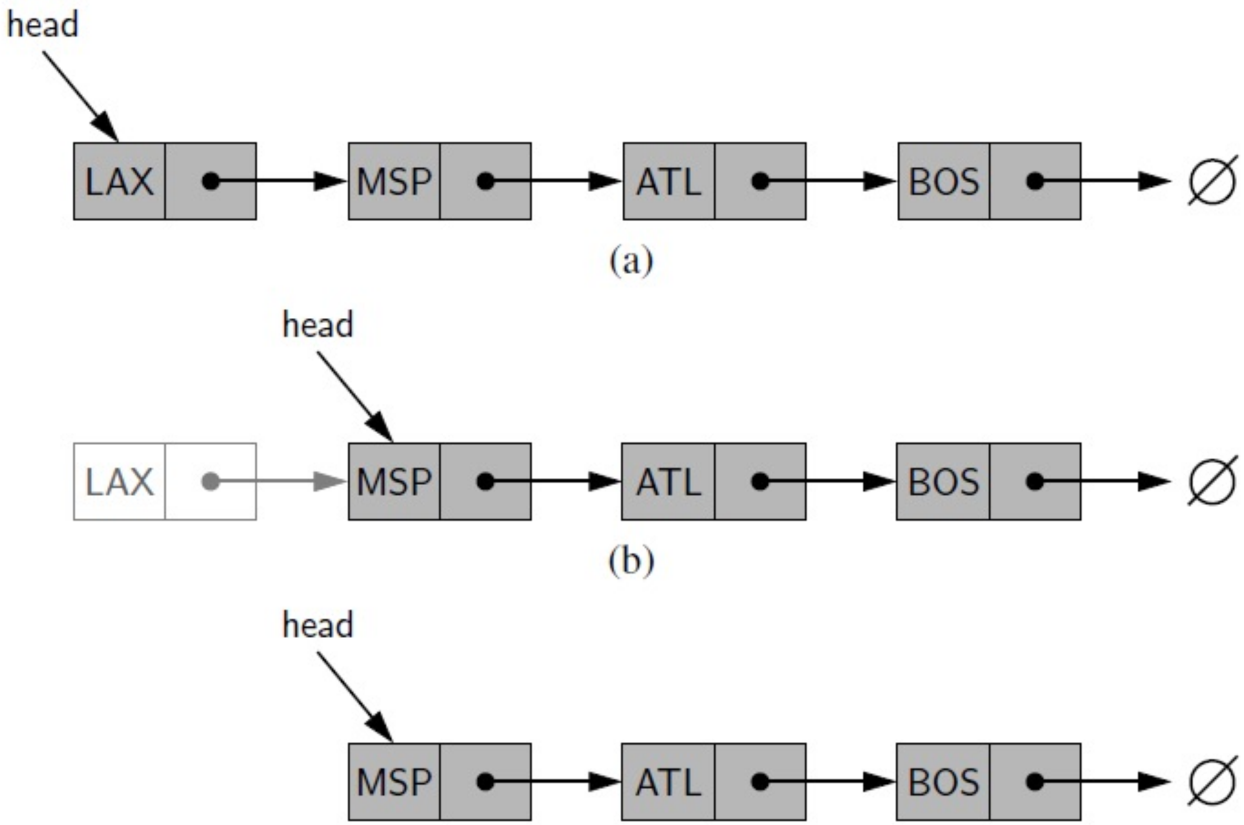

```python
def add_first(self, e):
    newest = Node(e, None)
    newest.pointer = self.head
    self.head = newest
    self.size = self.size+1

    if self.size == 1:
        self.tail = newest
```

INSERTING AN ELEMENT AT THE TAIL OF A SINGLY LINKED LIST




```
def add_last(self, e):
    newest = Node(e, None)
    if self.size > 0:
        self.tail.pointer = newest
    else:
        self.head = newest
    self.tail = newest
    self.size = self.size+1
```

REMOVING AN ELEMENT FROM THE HEAD OF A SINGLY LINKED LIST



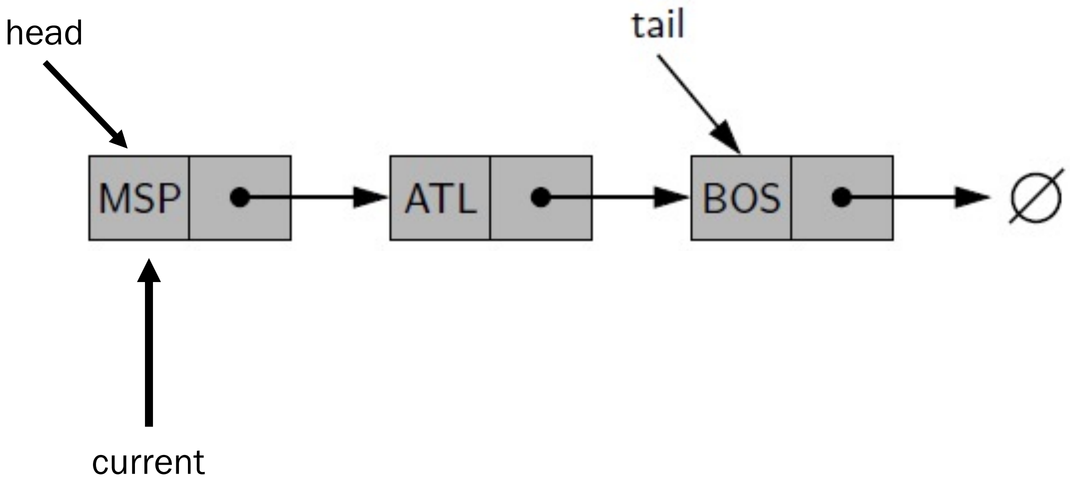

```python
def remove_first(self):
    if self.size == 0:
        print('The linked list is empty')
    elif self.size == 1:
        answer = self.head.element
        self.head = None
        self.tail = None
        self.size -= 1
        return answer
    else:
        answer = self.head.element
        self.head = self.head.pointer
        self.size = self.size - 1
        return answer
```

COMPLETE CODE OF LINKED LIST

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

class LinkedList:

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

    def add_first(self, e):
        newest = Node(e, None)
        newest.pointer = self.head
        self.head = newest
        self.size = self.size+1

        if self.size == 1:
            self.tail = newest
```

```python
    def add_last(self, e):
        newest = Node(e, None)
        if self.size > 0:
            self.tail.pointer = newest
        else:
            self.head = newest
        self.tail = newest
        self.size = self.size+1

    def remove_first(self):
        if self.size == 0:
            print('The linked list is empty')
        elif self.size == 1:
            answer = self.head.element
            self.head = None
            self.tail = None
            self.size -= 1
            return answer
        else:
            answer = self.head.element
            self.head = self.head.pointer
            self.size = self.size - 1
            return answer
```

REMOVING AN ELEMENT FROM THE TAIL OF A SINGLY LINKED LIST



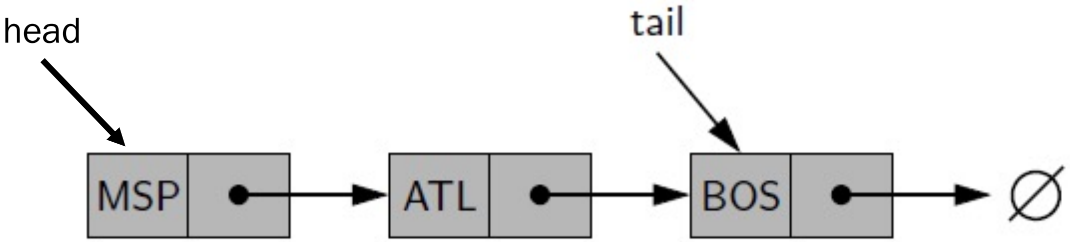

```
def remove_last(self):
    if self.size==0:
        print('The linked list is empty')
    elif self.size ==1:
        answer = self.tail.element
        self.head = None
        self.tail = None
        self.size = 0
        return answer
    else:
        answer = self.tail.element
        current = self.head
        while current.pointer != self.tail:
            current = current.pointer
        self.tail = current
        self.tail.pointer = None
        self.size = self.size - 1
        return answer
```

complexity: O(n)

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

COMPLEXITY COMPARISON

Methods	Compact Array	Singly Linked List
`add(e: E)`	$O(1)$	$O(1)$
`add(index: int, e: E)`	$O(n)$	$O(n)$
`clear()`	$O(1)$	$O(1)$
`contains(e: E)`	$O(n)$	$O(n)$
`get(index: int)`	$O(1)$	$O(n)$
`indexOf(e: E)`	$O(n)$	$O(n)$
`isEmpty()`	$O(1)$	$O(1)$
`lastIndexOf(e: E)`	$O(n)$	$O(n)$
`remove(e: E)`	$O(n)$	$O(n)$
`size()`	$O(1)$	$O(1)$
`remove(index: int)`	$O(n)$	$O(n)$
`set(index: int, e: E)`	$O(1)$	$O(n)$
`addFirst(e: E)`	$O(n)$	$O(1)$
`removeFirst()`	$O(n)$	$O(1)$



SOLVE PROBLEM WITH LINKEDLIST: SUB SEQUENCE

Given

$X = < x_1, x_2, \ldots, x_n>, n >= 0$

$Y = < y_1, y_2, \ldots, y_m>, m>= 0$

the function subsequence(X, Y) is true if and only if there exists a strictly increasing sequence of indices $K = < k_1, k_2, \ldots, k_n >$ such that every element x_i is equal to y_j where $j = k_i$

Consider X = < a b a > and Y = < b c a c b a>

subsequence(X,Y) is true because you can find a K = < 2 4 5>

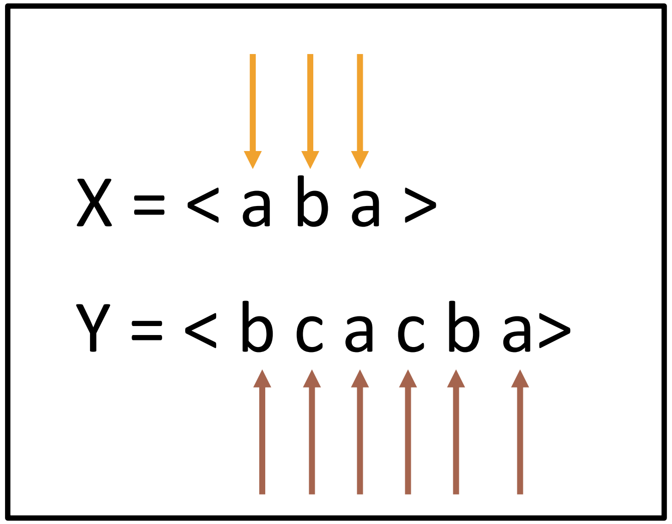
$x_0 = a$ is equal to $y_2 = a$

$x_1 = b$ is equal to $y_4 = b$

$x_2 = a$ is equal to $y_5 = a$

SOLUTION IDEAS

- Use iterator to iterate X
- For each element in X, iterate element in Y to see if there is a match
- Once match found, move the iterator for X forward
  - You may want to note the index for Y in this case
- Continue to move iterator for Y
- Loop until no more element in X or Y



PRACTICE: IMPLEMENT STACK WITH A SINGLY LINKED LIST

?

SOLUTION: 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

?

SOLUTION: IMPLEMENT QUEUE WITH A SINGLY LINKED LIST

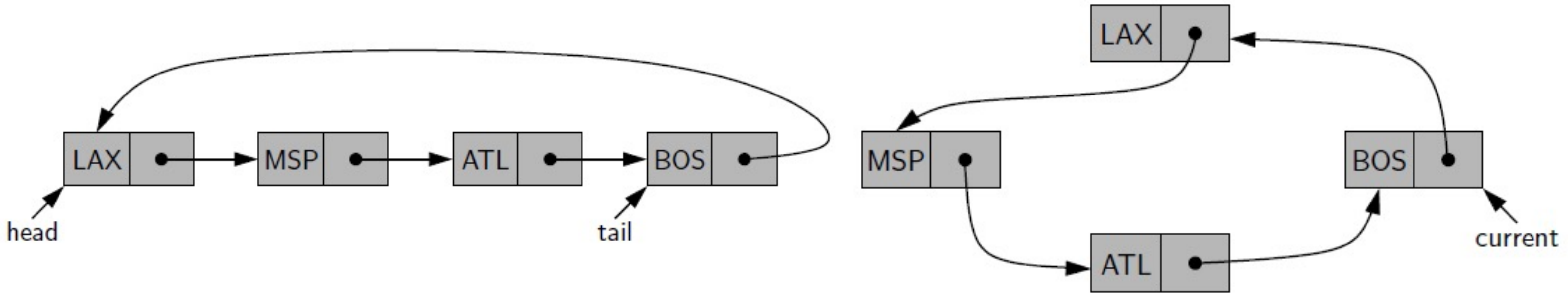
```
class LinkedQueue:

    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
Doubly Linked List

CIRCULARLY LINKED LIST

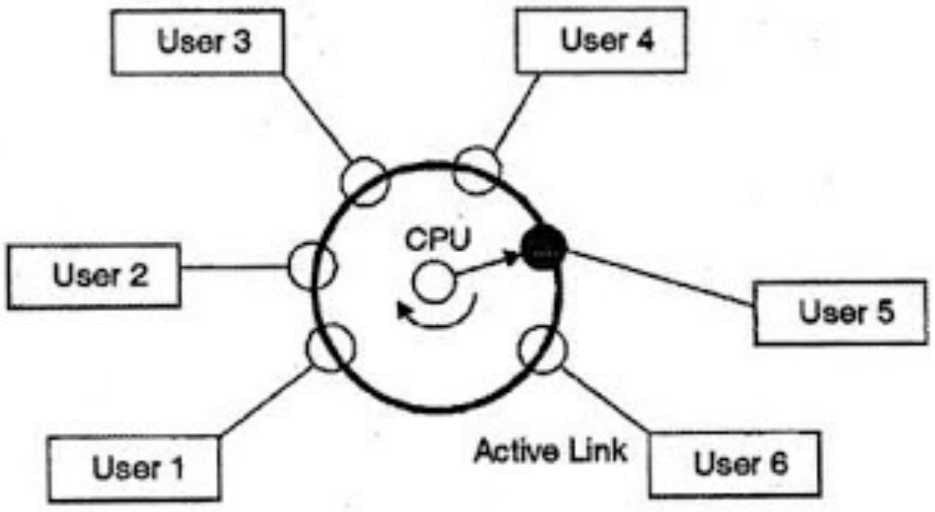
- The tail of a linked list can use its next reference to point back to the head of the list

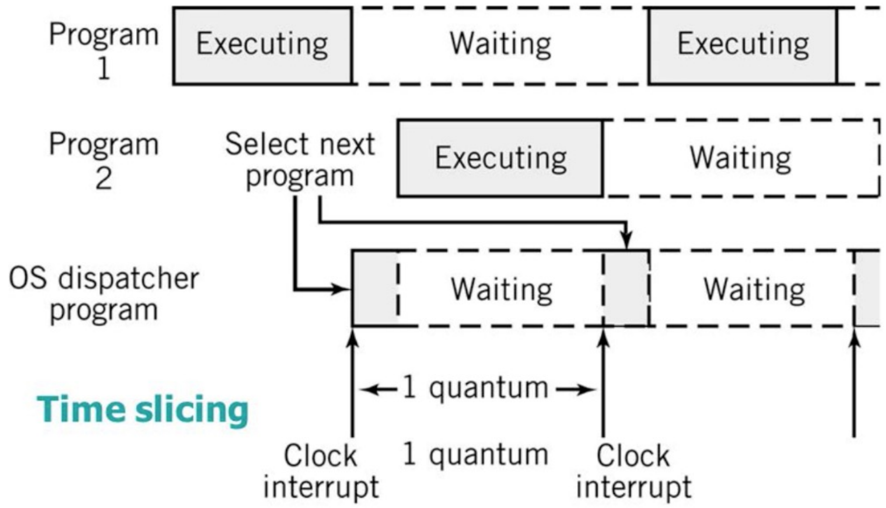


- Such a structure is usually called a circularly linked list

EXAMPLE: ROUND-ROBIN SCHEDULER

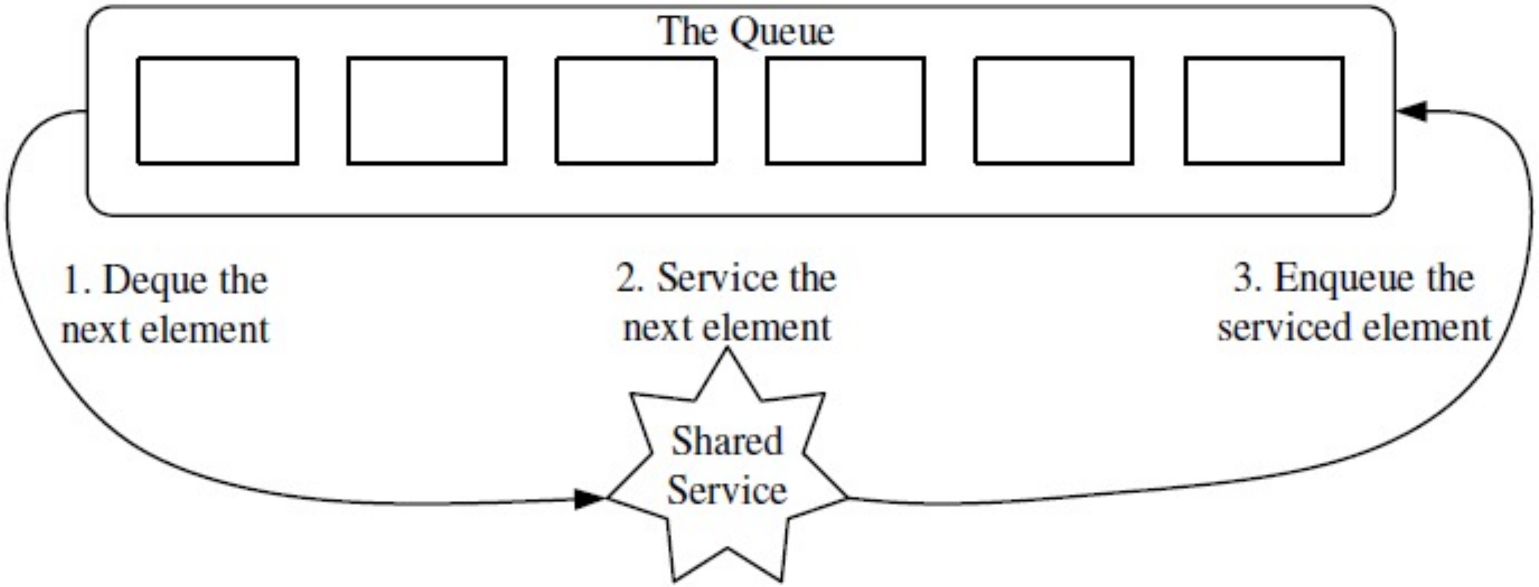
- Iterates through a collection of elements in a circular fashion and "serves" each element by performing a given action on it
- To fairly allocate a resource that must be shared by a collection of clients
  - Example: allocate slices of CPU time to various applications running concurrently on a computer





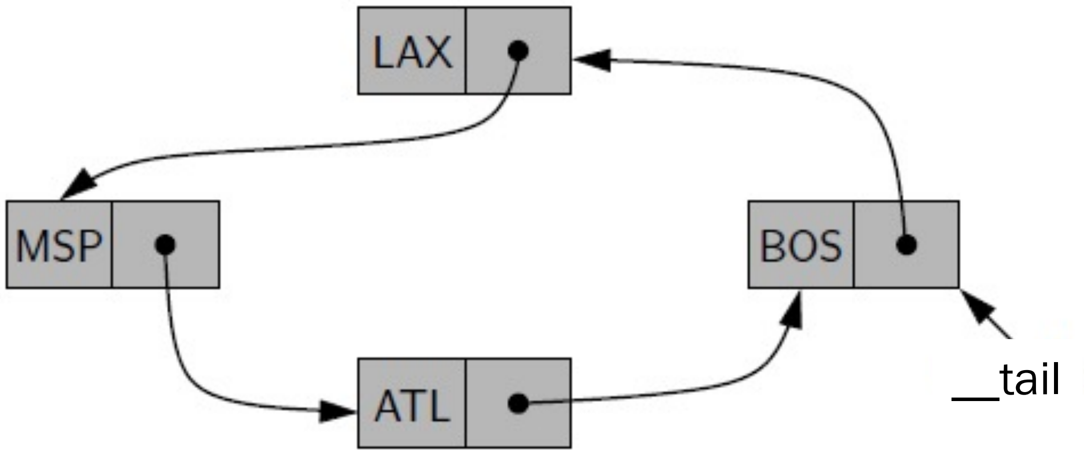
IMPLEMENTING ROUND-ROBIN SCHEDULER USING STANDARD QUEUE

- 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)



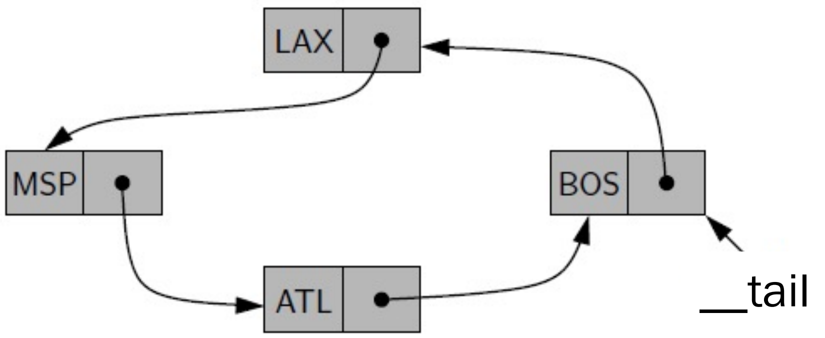
IMPLEMENT A QUEUE WITH A CIRCULARLY LINKED LIST

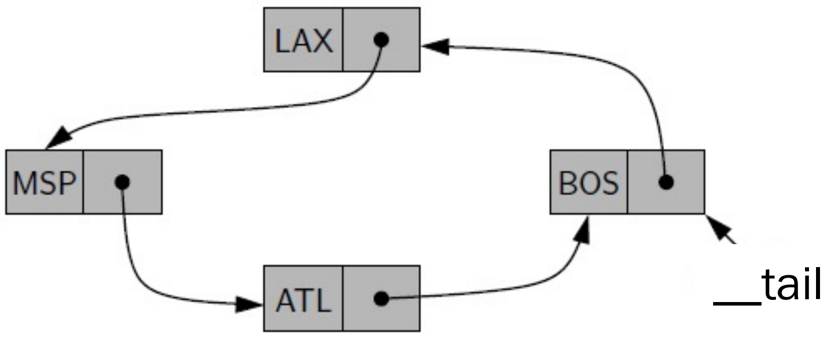
```python
class Node:
    def __init__(self, element, pointer):
        self.element = element
        self.pointer = pointer

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
```



IMPLEMENT A QUEUE WITH A CIRCULARLY LINKED LIST

```
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
```

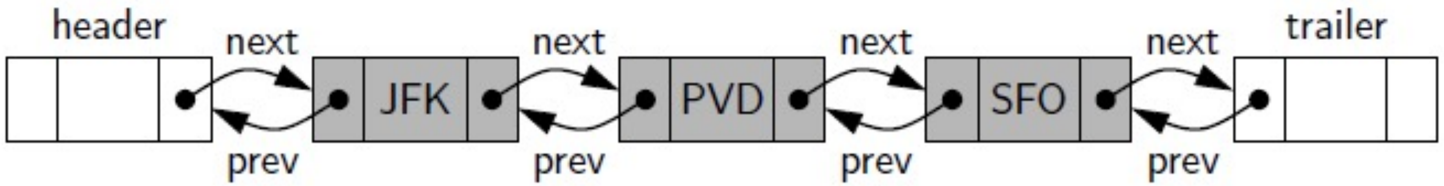




DOUBLY LINKED LIST

- Each node keeps an explicit reference to the node before it and a reference to the node after it

- Common Approach: add special nodes at both ends of the list

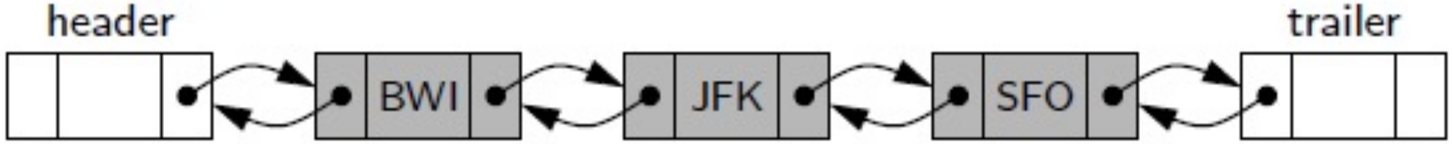


- These "dummy" nodes are known as sentinels (or guards), and they do not store elements of the primary sequence
- To avoid some special cases when operating near the boundaries

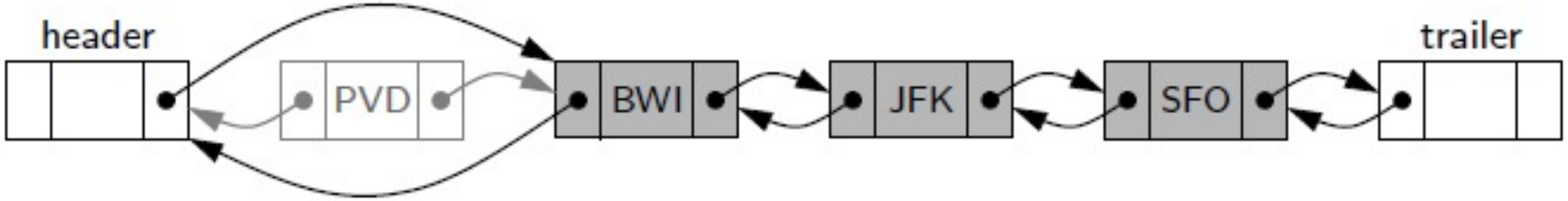
CODE FOR THE DOUBLY LINKED LIST

```
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
```

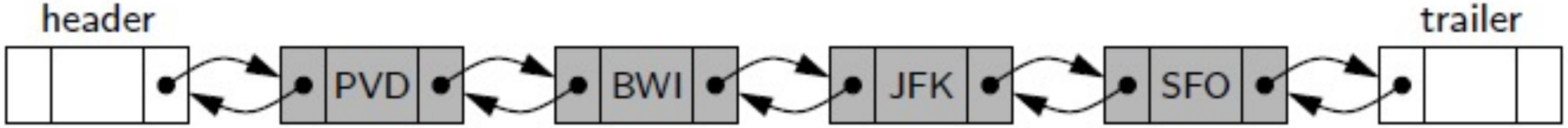
INSERTING AT THE HEAD OF THE DOUBLY LINKED LIST



(a)

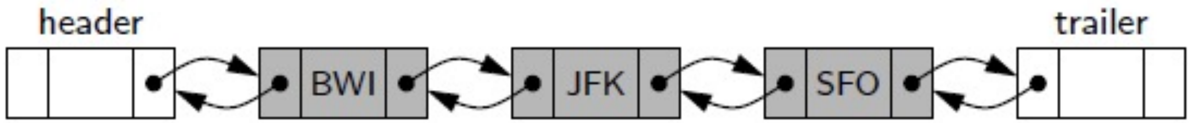


(b)

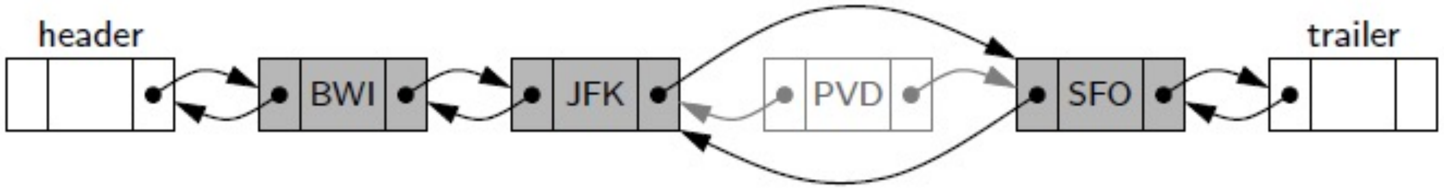


(c)

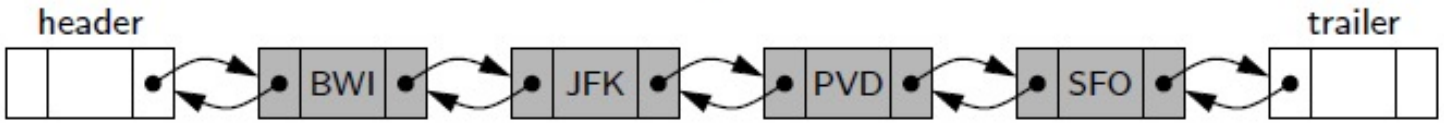
INSERTING IN THE MIDDLE OF A DOUBLY LINKED LIST



(a)



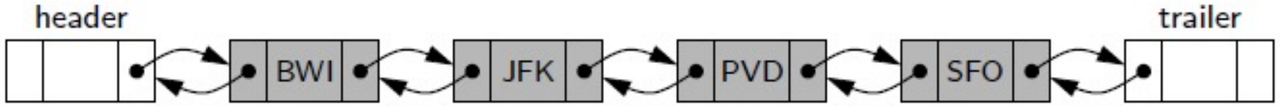
(b)



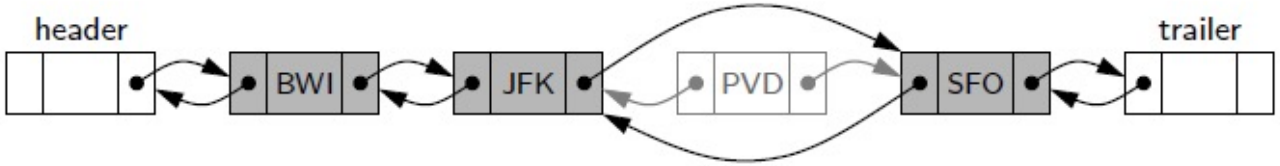
(c)

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

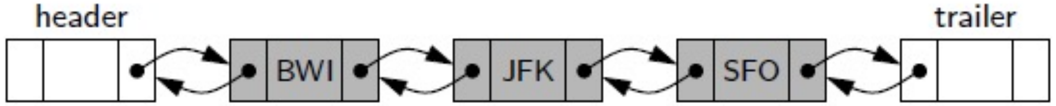
DELETING FROM THE DOUBLY LINKED LIST



(a)



(b)



(c)

```
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
```

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
```

```
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
```

CODE FOR THE DOUBLY LINKED LIST

```
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()
```

Sorting Algorithms

SORTING ALGORITHMS

- Selection Sort
- Bubble Sort
- Quick Sort
- …

Default Assumption: from smallest to largest

SELECTION SORT

- Basic idea:
  - Find the smallest value in list and move to index 0
  - Find the second smallest value and move to index 1
  - Find the third smallest value and move to index 2
  - …

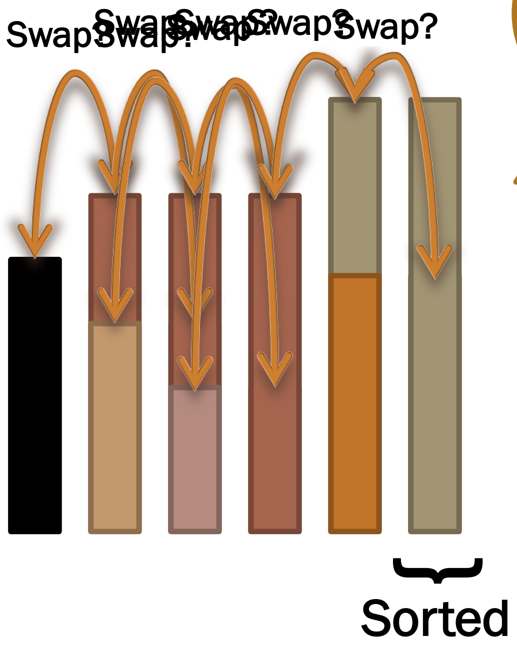
0	1	2	3	4
13	17	6	2	34
13	17	6	2	34
2	17	6	13	34
2	6	17	13	34
2	6	13	17	34

BUBBLE SORT

- A simple sorting algorithm
- Its general procedure is:
  - Iterate over the list, compare every element i with the following element i+1, and swap them if i is larger
  - Iterate over the list again and repeat the procedure in step 1, but ignore the last element in the list
  - 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

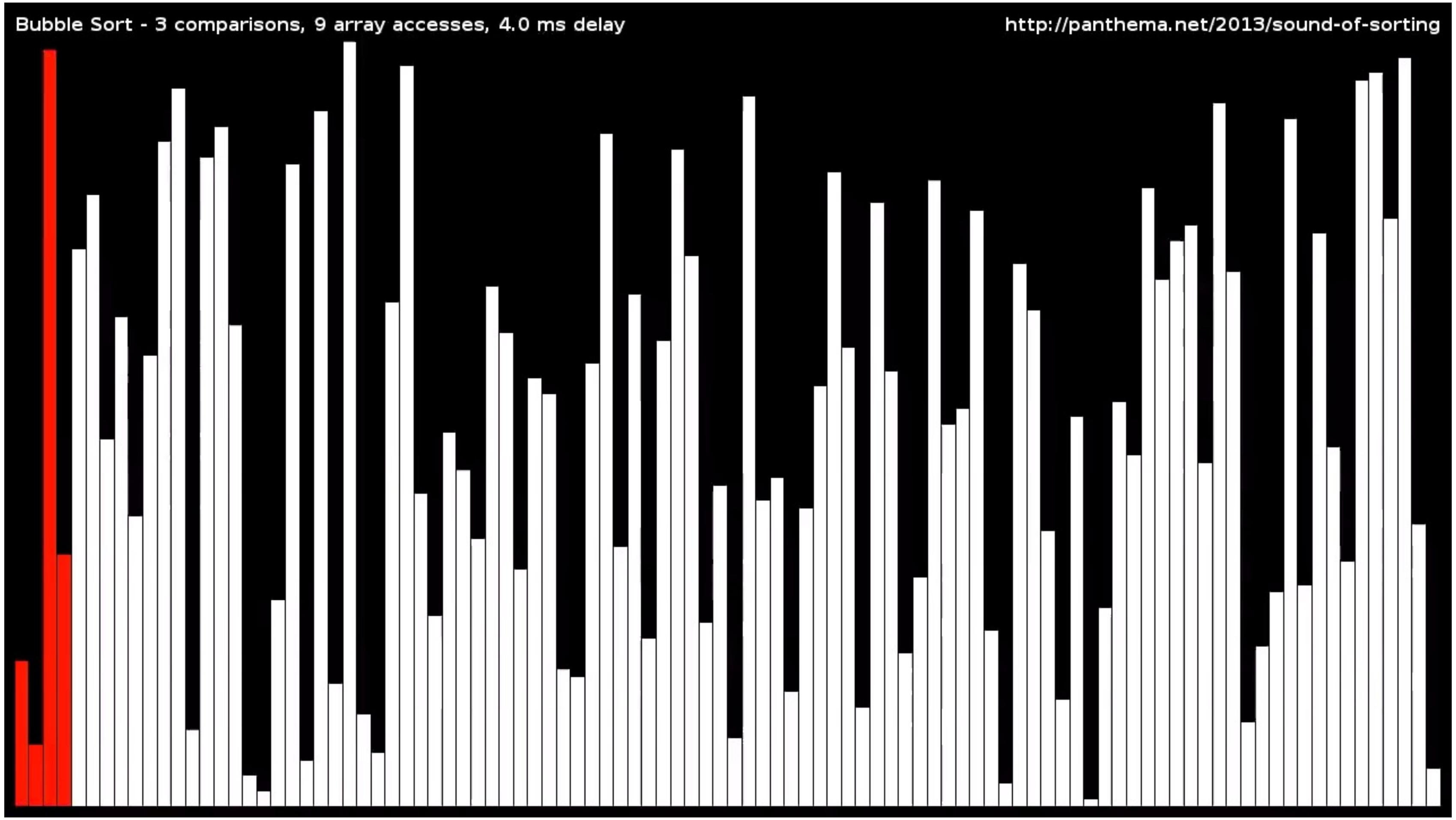
BUBBLE SORT

- Example: Sorting by length (increasing)



How many iterations are needed for complete sort?

BUBBLE SORT ILLUSTRATION



PRACTICE: BUBBLE SORT OVER A STANDARD LIST

```python
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:

```
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
```

Change the element value

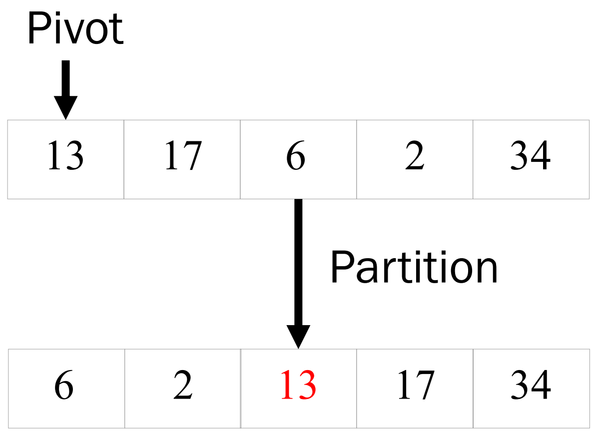
```
def outputQ(q):
    pointer = q.head

    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)
```

ANOTHER SOLUTION

```
def linkedBubble(linkedlist):
    listLength = linkedlist.size
    while listLength > 0:
        listLength -= 1
        # the first one
        if linkedlist.head.element > linkedlist.head.pointer.element:
            currentHead = linkedlist.head
            linkedlist.head = linkedlist.head.pointer
            currentHead.pointer = linkedlist.head.pointer
            linkedlist.head.pointer = currentHead
        # from the second one
        current = linkedlist.head
        while current.pointer.pointer != None:
            if current.pointer.element > current.pointer.pointer.element:
                temp = current.pointer
                current.pointer = temp.pointer
                temp.pointer = current.pointer.pointer
                current.pointer.pointer = temp
            current = current.pointer
    return linkedlist
```

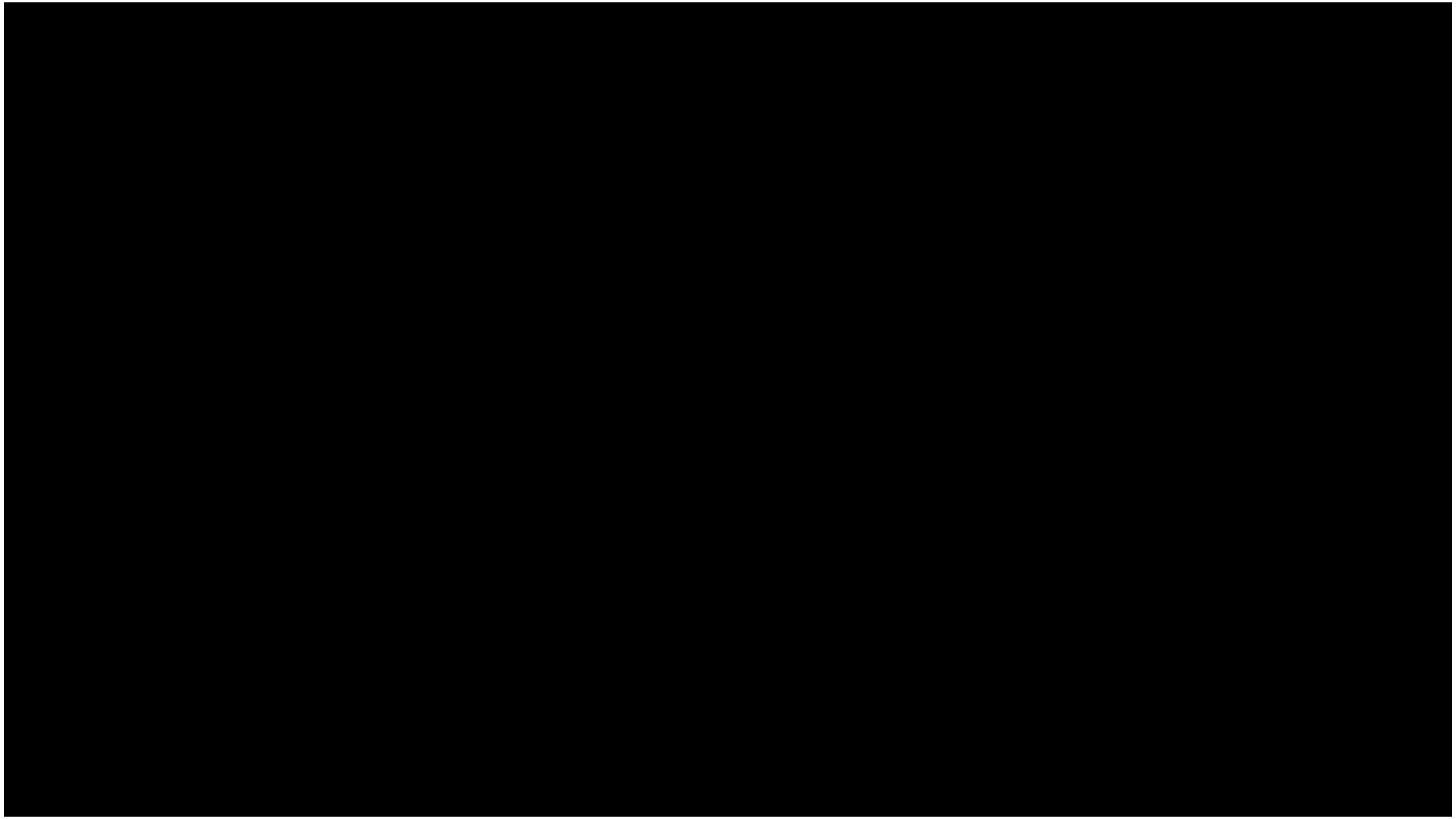
Change the node order

QUICK SORT

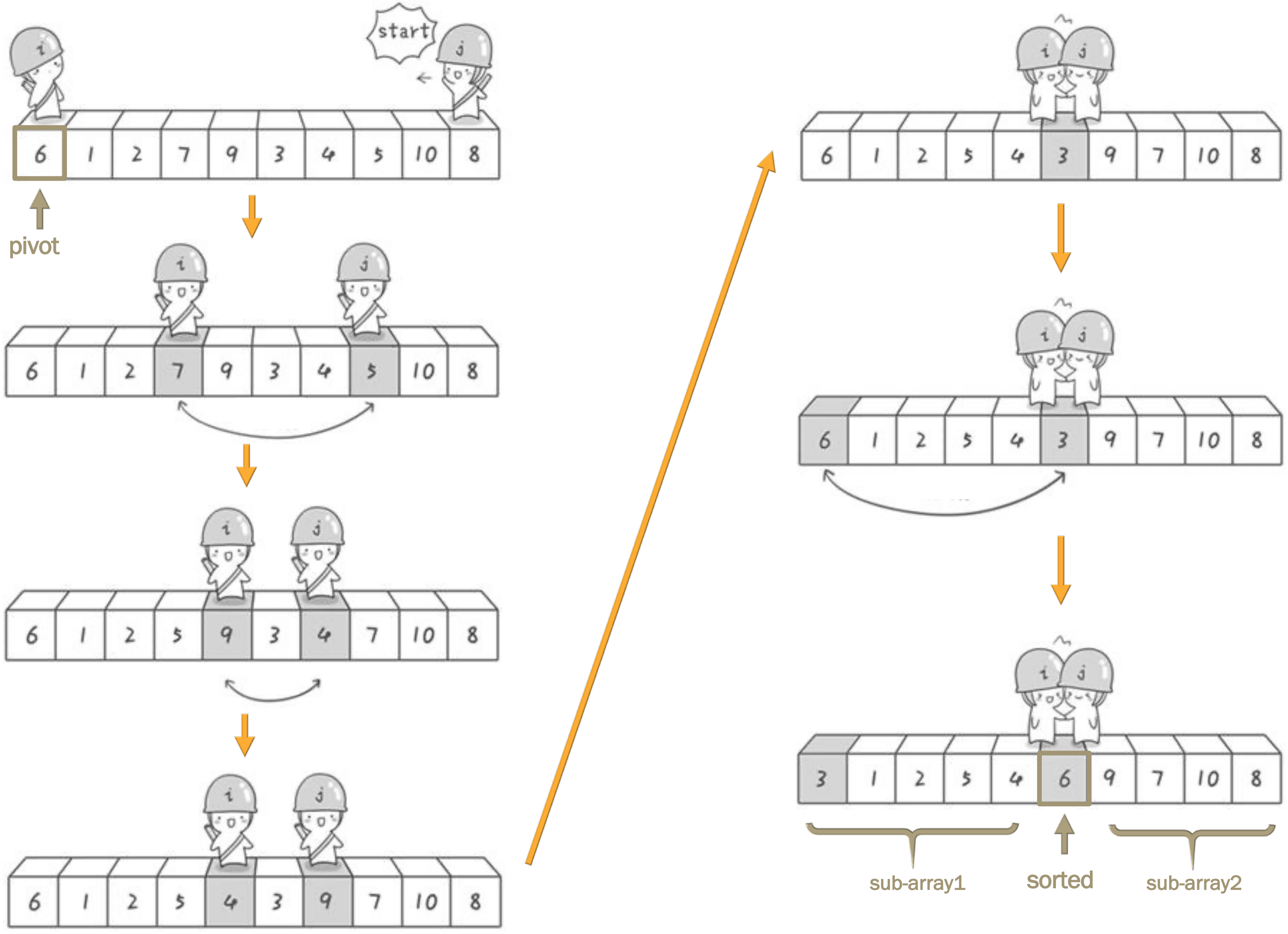


- a widely used algorithm
- more efficient than bubble sort
- Idea:
  - Pick an element, called a pivot, from the list
  - Partition: Reorder the array
    - elements with values less than pivot come before the pivot
    - elements with values greater than pivot come after it
    - equal values can go either way
    - the pivot is in its final position after this partitioning
    - creates two sub-list
  - Recursively apply the above steps to the sub-list

QUICK SORT WITH HUNGARIAN FOLK DANCE



PARTITION ILLUSTRATION



PRACTICE: QUICK SORT OVER A STANDARD LIST

?

PRACTICE: QUICK SORT OVER A STANDARD LIST

```python
def partition(array, start, end):
    pivot = array[start]
    low = start + 1
    high = end

    while True:
        while low <= high and array[high] >= pivot:
            high = high - 1
        while low <= high and array[low] <= pivot:
            low = low + 1

        if low <= high:
            array[low], array[high] = array[high], array[low]
        else
            break

    array[start], array[high] = array[high], array[start]

return high
```

PRACTICE: QUICK SORT OVER A STANDARD LIST

```python
def quick_sort(array, start, end):
    if start < end:
        p = partition(array, start, end)
        quick_sort(array, start, p-1)
        quick_sort(array, p+1, end)


def main():
    array = [29,99,27,41,66,28,44,78,84]
    quick_sort(array, 0, len(array) - 1)
    print(array)
```

PRACTICE: QUICK SORT OVER A SINGLY LINKED LIST

?