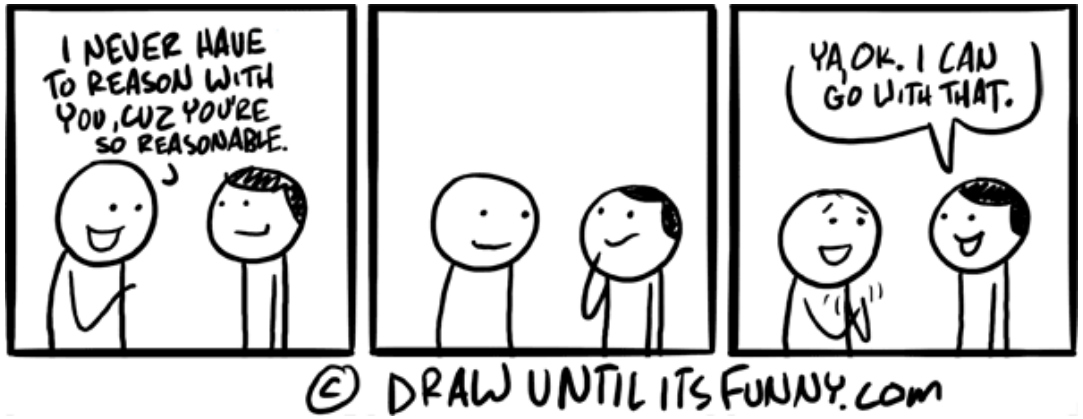
LINKEDLIST

School of Artificial Intelligence

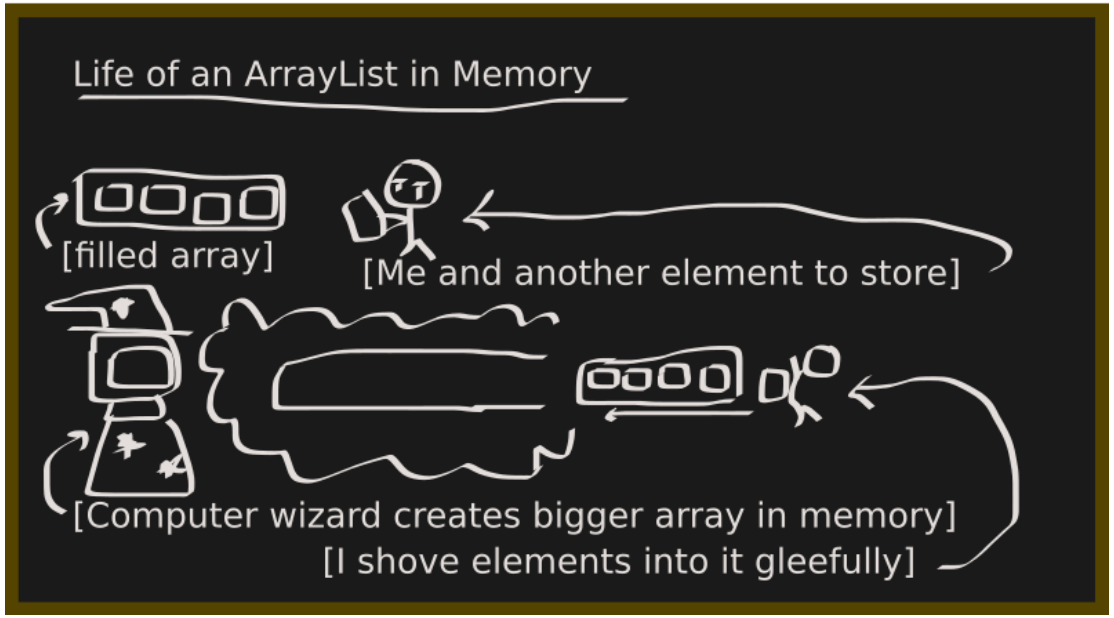
3RS

- Relax
- Reasonable
- Respect



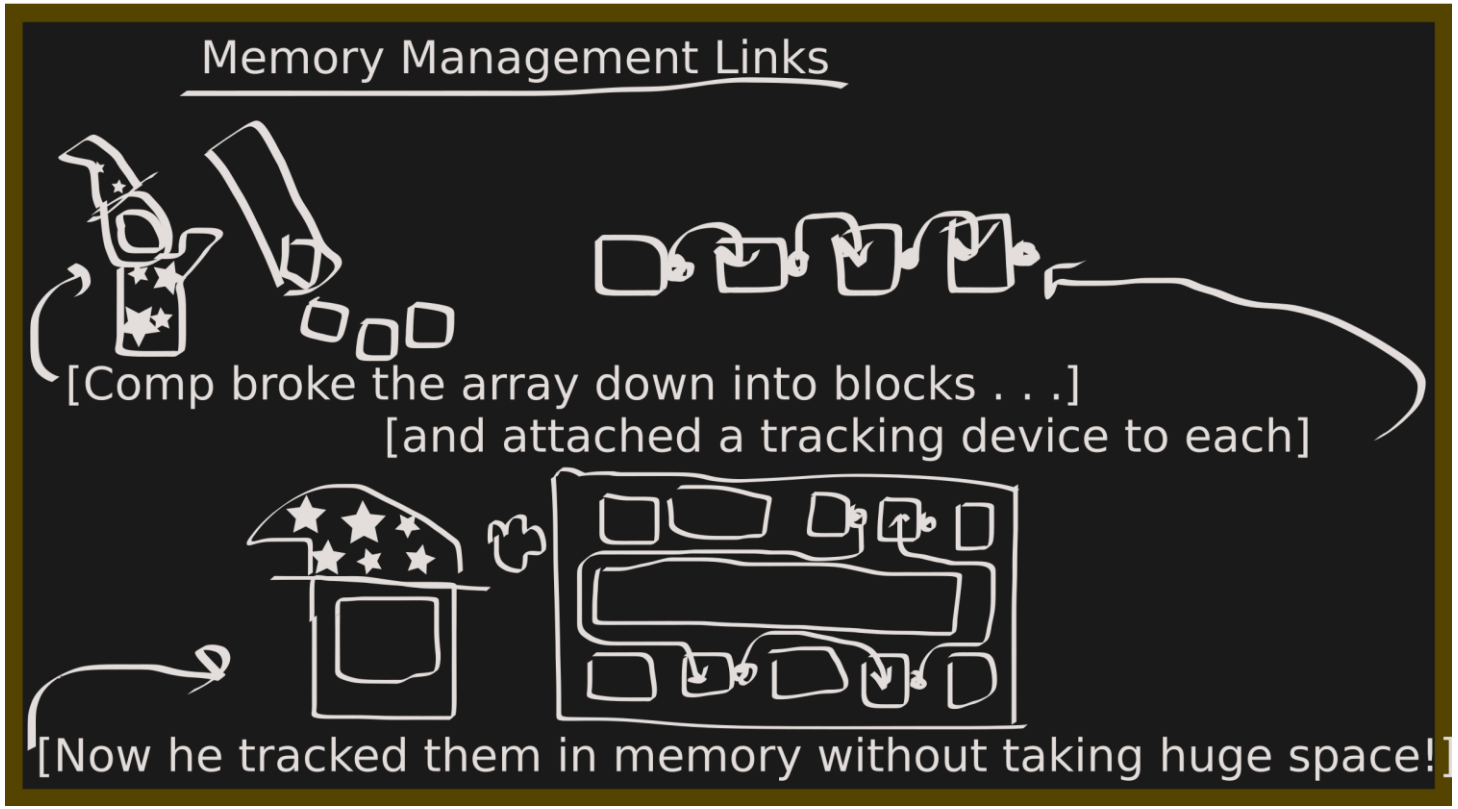
PREVIOUSLY ON DS&A

- Array
  - Referential Array
  - Compact Array
- Sorting
  - Insertion Sort
  - Merge Sort
- Stack
- Queue
  - Implemented with circular array
- Deque
  - Double ended queue



UP NEXT

- LinkedList (链表)
- Singly Linked Lists
- Circularly Linked Lists
- Doubly Linked Lists

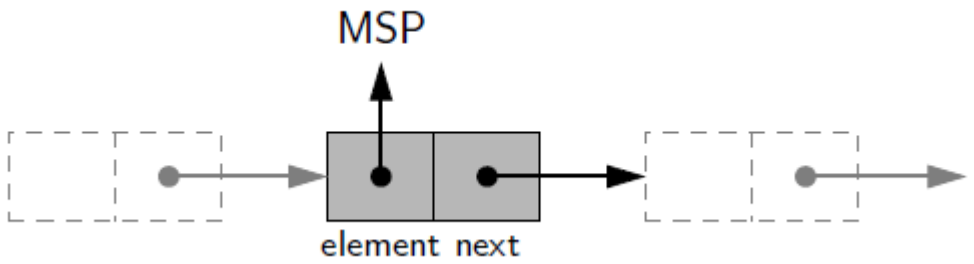


LINKED LISTS (链表)

- Disadvantages of a dynamic array
  - The length of a dynamic array might be longer than the actual number of elements that it stores
  - Amortised bounds for operations may be unacceptable in real-time systems
  - Insertions and deletions at interior positions of an array are expensive
- Linkedlist
  - Stores values as an array does
  - Distributed representation
    - A lightweight object, known as a **node**, is allocated for each element
  - **Node** maintains a reference to its element + reference to its neighbouring **node**

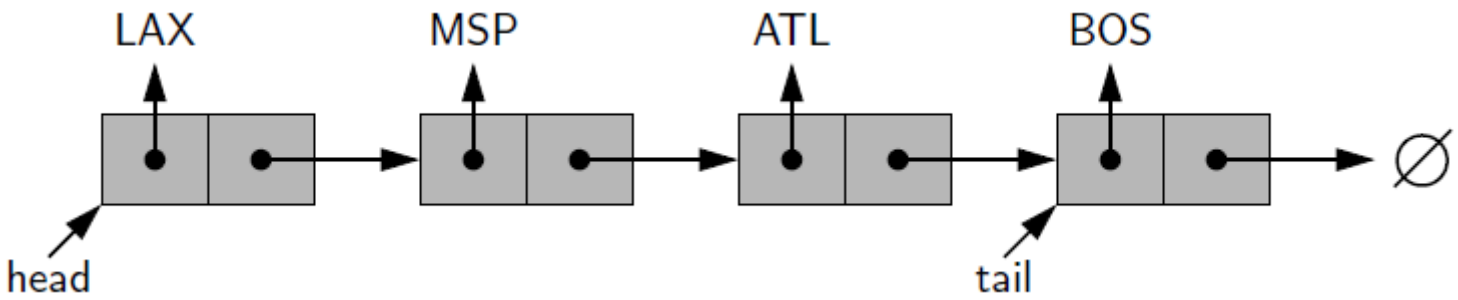
SINGLY LINKED LISTS (单链表)

- Singly linked list
  - Collection of nodes that form a linear sequence
  - Each node:
    1. Reference to an object
    2. Reference to the *next* node



SINGLY LINKED LISTS (单链表)

- Head（首）: first node in the linked list
- Tail（尾）: last node in the linked list
  - None as its *next* reference
- Linkedlist traversing (遍历):
  - Starting at the head and following the *next* reference until the tail
  - Often referred to as *link hopping* or *pointer hopping*

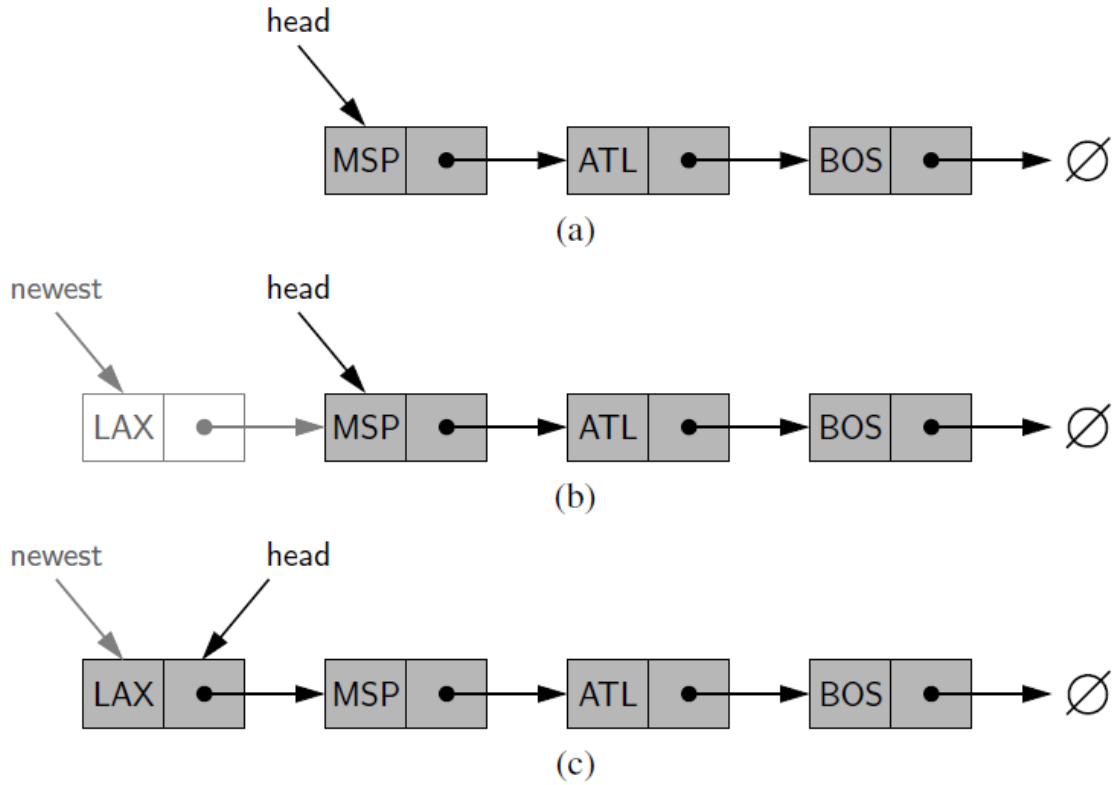


SINGLY LINKED LISTS (单链表)

- In memory representation
  - Each node as a unique object
    - Reference to its element (value)
    - Reference to the *next* node (or None)
  - An object maintains the linkedlist
    - Reference to the head of the list – Why?
    - Reference to the tail of the list – why?
    - Number of elements in the linkedlists

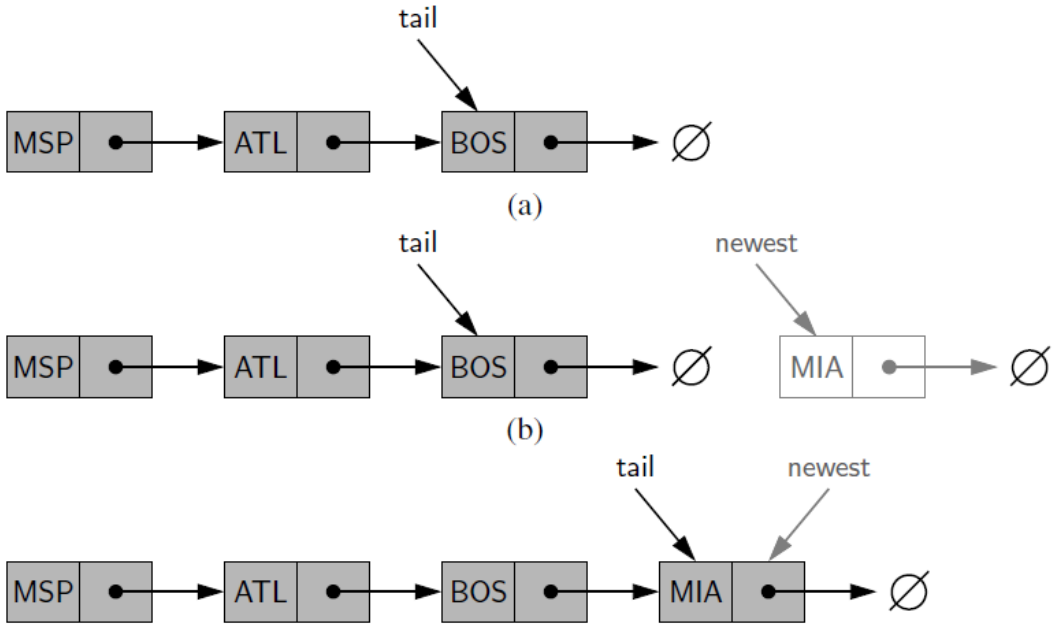
SINGLY LINKED LISTS (单链表)

- Inserting an element at the head of a singly linked list
  1. Create a new node, with its value
  2. Set the *next* reference of the node to the head of the linkedlist
  3. Update the linkedlists' head
  4. Update the linkedlists' size



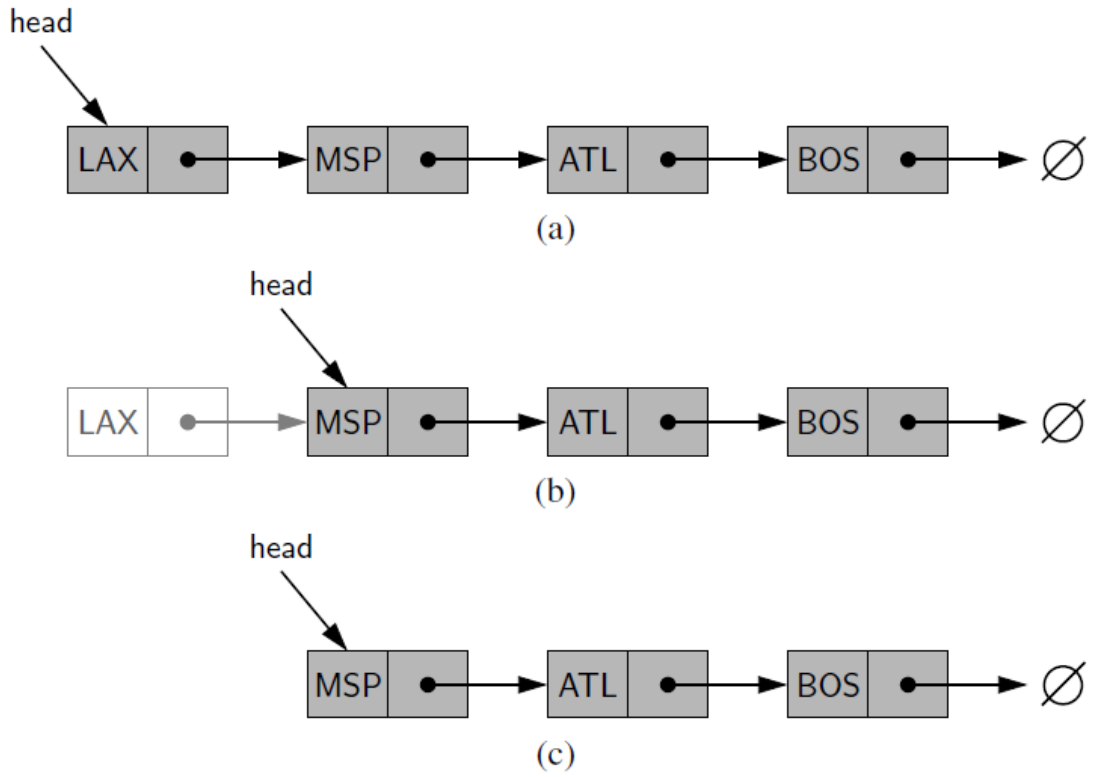
SINGLY LINKED LISTS (单链表)

- Inserting an element at the tail of a singly linked list
  1. Create a new node, with its value
  2. Set the *next* reference of the node to None
  3. Set the *next* reference of the tail of the linkedlist to the new node
  4. Update the linkedlists' tail
  5. Update the linkedlists' size



SINGLY LINKED LISTS (单链表)

- Removing an element from the head of a singly linked list
  1. If L is empty, raise an error
  2. L.head = L.head.next
  3. L.size = L.size - 1
- What about removing the tail of a singly linked list?
  - Problems?
  - A traversal through the entire linkedlist is necessary with O(n)
  - How do we solve this problem?



STACK IMPLEMENTED WITH A SINGLY LINKED LIST

- Abstract Data Type (抽象数据类 - ADT) of a Stack
  - S.push(e): adds e to the top of stack S
  - S.pop(): removes and return the top element from stack S
    - Error when stack is empty
  - S.top(): returns a reference to the top element of stack S (but not to remove it)
  - S.is_empty(): returns True if S does not have any elements
  - len(s): returns the number of elements in S
- Question: with a linkedlist, which side should be the top of the stack?

STACK IMPLEMENTED WITH A SINGLY LINKED LIST

- Node class as the fundamental of the linkedlist
  - Element: to store the value of the node
  - Next: to store the reference to the next node
- The implementation of the Stack with linkedlist
  - push()
  - pop()
  - top()

STACK IMPLEMENTED WITH A SINGLY LINKED LIST

- Node class as the fundamental of the linkedlist
  - Element: to store the value of the node
  - Next: to store the reference to the next node
- The implementation of the Stack with linkedlist
  - push()
  - pop() – raise error when stack is empty
  - top() – raise error when stack is empty

STACK IMPLEMENTED WITH A SINGLY LINKED LIST

- Efficiency of Stack operations with LinkedList implementation?
  - S.push()
  - S.pop()
  - S.top()
  - Len(S)
  - S.is_empty()
  - All O(1)

QUEUE IMPLEMENTED WITH A SINGLY LINKED LIST

- Abstract Data Type (ADT)
  - Q.enqueuer(e): add element e to the back of queue Q （入队）
  - Q.dequeue(): remove and return the first element from queue Q （出队）
  - Q.first(): returns a reference to the element at the front of queue Q
    - Error if Q is empty
  - Q.is_empty(): returns True if queue Q does not contain any elements
  - len(Q): returns the number of elements in Q
- Question: which side of the queue should be the head of a linkedlist?
  - Front of the queue as the head of the linkedlist

QUEUE IMPLEMENTED WITH A SINGLY LINKED LIST

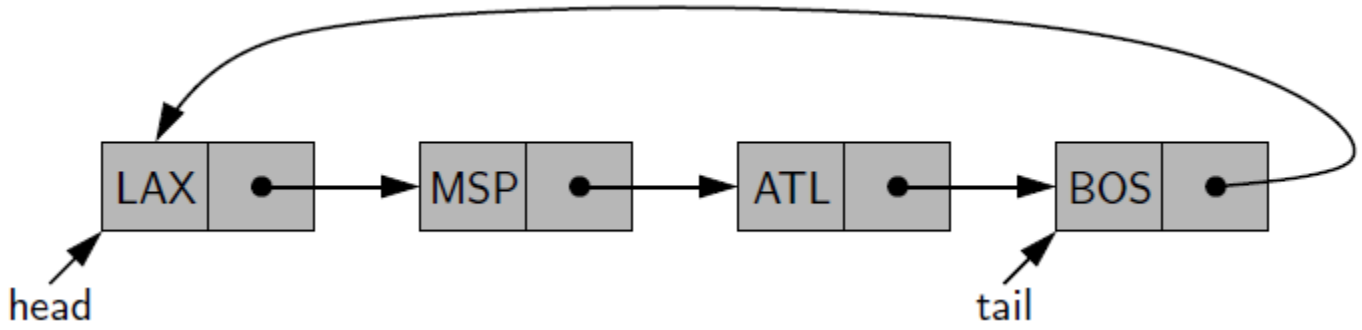
```
1  class LinkedQueue:
2    """FIFO queue implementation using a singly linked list for storage."""
3
4    class _Node:
5      """Lightweight, nonpublic class for storing a singly linked node."""
6      (omitted here; identical to that of LinkedStack._Node)
7
8    def __init__(self):
9      """Create an empty queue."""
10     self._head = None
11     self._tail = None
12     self._size = 0                          # number of queue elements
13
14   def __len__(self):
15     """Return the number of elements in the queue."""
16     return self._size
17
18   def is_empty(self):
19     """Return True if the queue is empty."""
20     return self._size == 0
21
22   def first(self):
23     """Return (but do not remove) the element at the front of the queue."""
24     if self.is_empty():
25       raise Empty('Queue is empty')
26     return self._head._element              # front aligned with head of list
27   def dequeue(self):
28     """Remove and return the first element of the queue (i.e., FIFO).
29
30     Raise Empty exception if the queue is empty.
31     """
32     if self.is_empty():
33       raise Empty('Queue is empty')
34     answer = self._head._element
35     self._head = self._head._next
36     self._size -= 1
37     if self.is_empty():                     # special case as queue is empty
38       self._tail = None                     # removed head had been the tail
39     return answer
40
41   def enqueue(self, e):
42     """Add an element to the back of queue."""
43     newest = self._Node(e, None)            # node will be new tail node
44     if self.is_empty():
45       self._head = newest                   # special case: previously empty
46     else:
47       self._tail._next = newest
48     self._tail = newest                     # update reference to tail node
49     self._size += 1
```

QUEUE IMPLEMENTED WITH A SINGLY LINKED LIST

- Efficiency of Stack operations with LinkedList implementation?
  - Q.enqueue(e)
  - Q.dequeuer()
  - Q.first()
  - Q.is_empty()
  - len(Q)

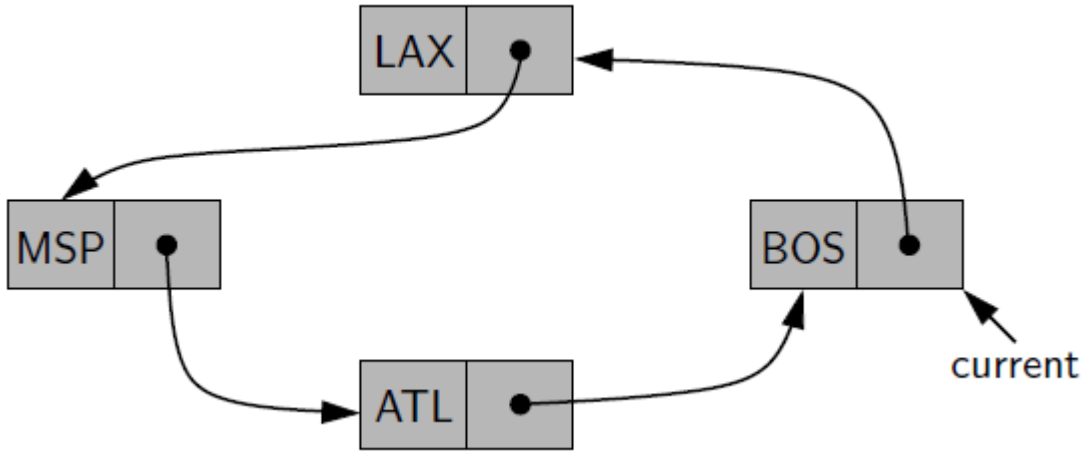
CIRCULARLY LINKEDLISTS（环形链表）

- Last week we talked about “circular” array
- We have seen that circular array is artificial
  - Need to maintain the reference to the front of the array
  - Need to compute the available slot of the array (tail)
  - Need to grow/shrink the array when necessary
- Circular linked list
  - Tail’s *next* refers to head



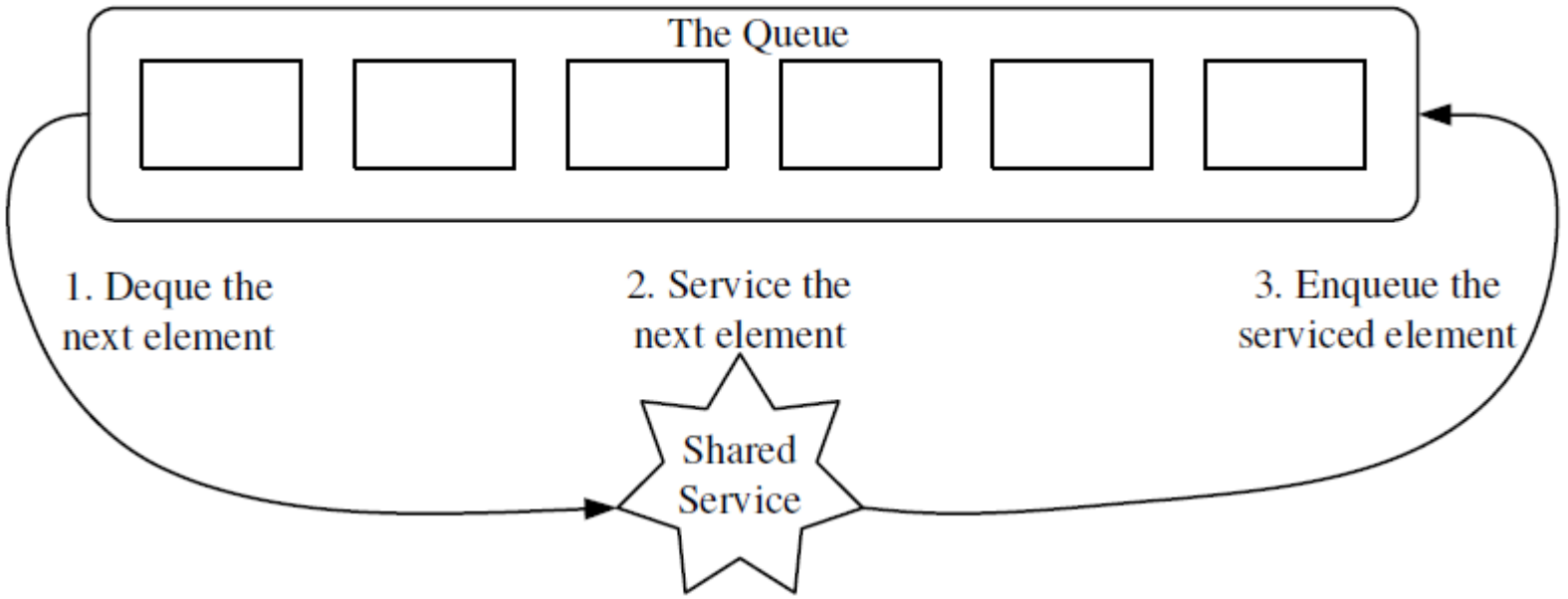
CIRCULARLY LINKEDLISTS（环形链表）

- Often used to represent cyclic data sets
  - No clear notion of a beginning and an end
  - Circular underground (地铁)
  - The order which players take turns during a game
- As data structure designers
  - Must maintain a reference to a particular node to make use of the list
  - We use **current**



ROUND-ROBIN SCHEDULERS
（轮询调度器）

- Example of a circularly linked list
- Iterates through a collection of elements in a circular fashion
- “serves” each element by performing a given action on it
- Used to allocate resource that must be shared by a collection of clients (processes)
  - E.g. CPU time
  1. e = Q.dequeue()
  2. Serve element e
  3. Q.enqueue(e)



ROUND-ROBIN SCHEDULERS（轮询调度器）

- Implemented with a circularly linked list

```
1  class CircularQueue:
2    """Queue implementation using circularly linked list for storage."""
3
4    class _Node:
5      """Lightweight, nonpublic class for storing a singly linked node."""
6      (omitted here; identical to that of LinkedStack._Node)
7
8    def __init__(self):
9      """Create an empty queue."""
10     self._tail = None                    # will represent tail of queue
11     self._size = 0                       # number of queue elements
12
13   def __len__(self):
14     """Return the number of elements in the queue."""
15     return self._size
16
17   def is_empty(self):
18     """Return True if the queue is empty."""
19     return self._size == 0
```

ROUND-ROBIN SCHEDULERS（轮询调度器）

- Implemented with a circularly linked list

```
20    def first(self):
21      """Return (but do not remove) the element at the front of the queue.
22
23      Raise Empty exception if the queue is empty.
24      """
25      if self.is_empty( ):
26        raise Empty('Queue is empty')
27      head = self._tail._next
28      return head._element
29
30    def dequeue(self):
31      """Remove and return the first element of the queue (i.e., FIFO).
32
33      Raise Empty exception if the queue is empty.
34      """
35      if self.is_empty( ):
36        raise Empty('Queue is empty')
37      oldhead = self._tail._next
38      if self._size == 1:                    # removing only element
39        self._tail = None                    # queue becomes empty
40      else:
41        self._tail._next = oldhead._next     # bypass the old head
42      self._size -= 1
43      return oldhead._element
```

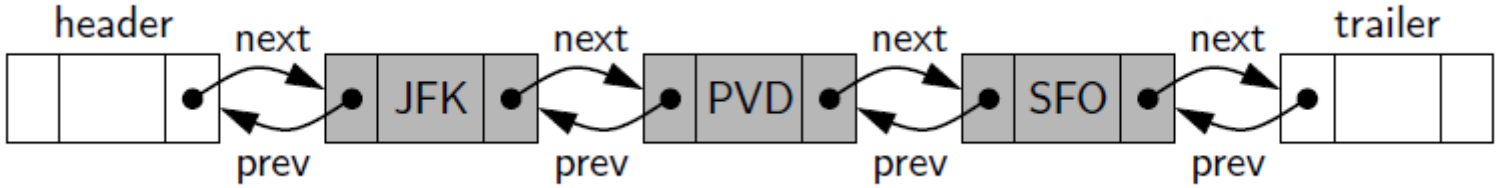
ROUND-ROBIN SCHEDULERS
（轮询调度器）

- Implemented with a circularly linked list

```
45    def enqueue(self, e):
46      """Add an element to the back of queue."""
47      newest = self._Node(e, None)          # node will be new tail node
48      if self.is_empty( ):
49        newest._next = newest               # initialize circularly
50      else:
51        newest._next = self._tail._next     # new node points to head
52        self._tail._next = newest           # old tail points to new node
53      self._tail = newest                   # new node becomes the tail
54      self._size += 1
55
56    def rotate(self):
57      """Rotate front element to the back of the queue."""
58      if self._size > 0:
59        self._tail = self._tail._next       # old head becomes new tail
```

DOUBLY LINKED LISTS（双链表）

- Limitations on Singly linked list
  - Delete the tail of the linked list is inefficient – always O(n)
- Doubly linkedlist
  - Node – reference to both *previous* and *next* of it
  - Doubly linked list
- Structure: similar to singly linked list, with one exception
  - Head and trailer sentinels
  - A header node and a trailer node: empty 'dummy' nodes （哑元节点）
    - Also known as **sentinels** (哨兵节点)



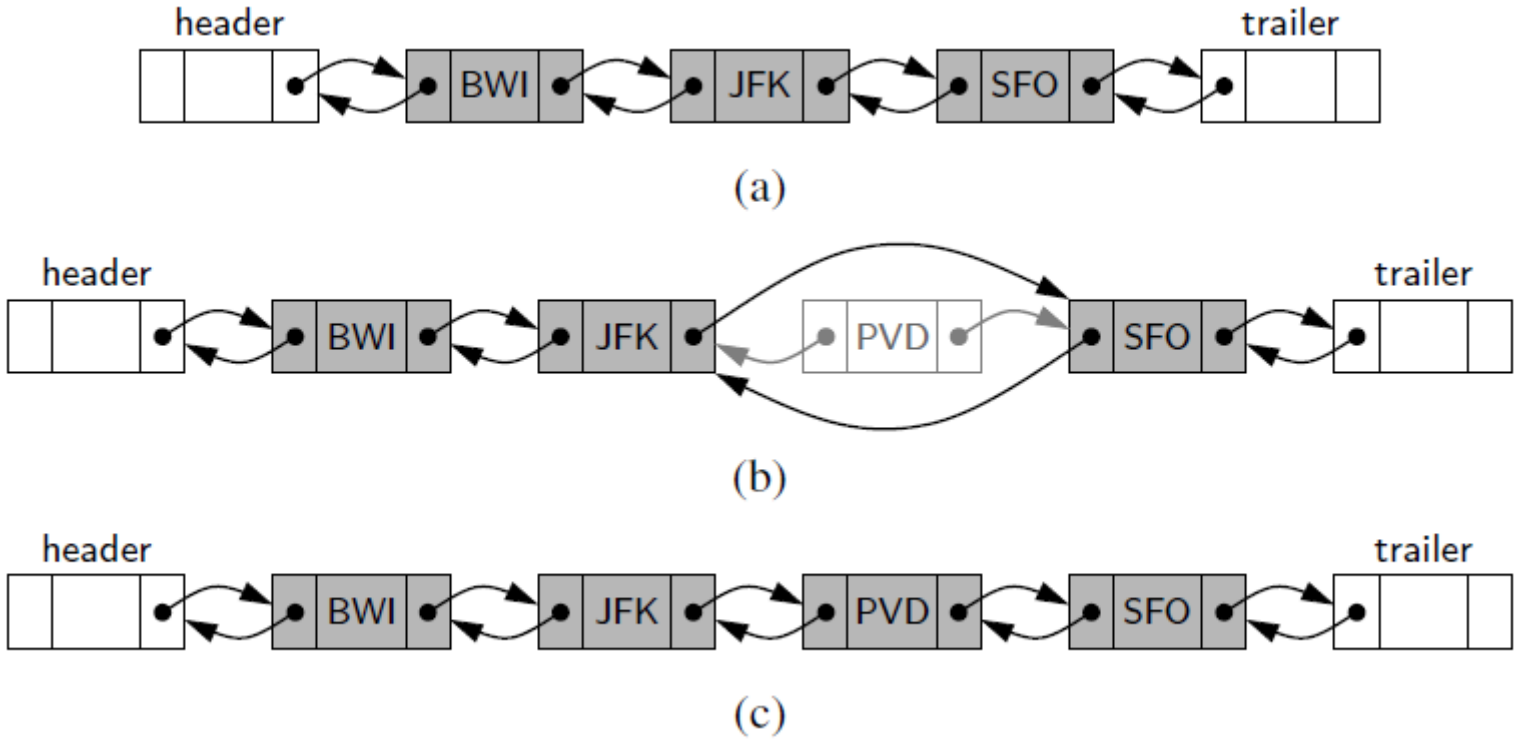
DOUBLY LINKED LISTS（双链表）

- Advantages of using sentinels
  - Header and trailer never change, only the nodes between them change
  - All insertions can be treated in a unified manner
    - A node is always placed between a pair of existing nodes
  - All deletions can also be treated in a unified manner
  - There is no special care for cases when the list is empty

```
new_node = new Node(e, None)
if(self.is_empty()) {
    self_head  = new_node
}
Else {
    self._tail._next = new_node
}
…
```

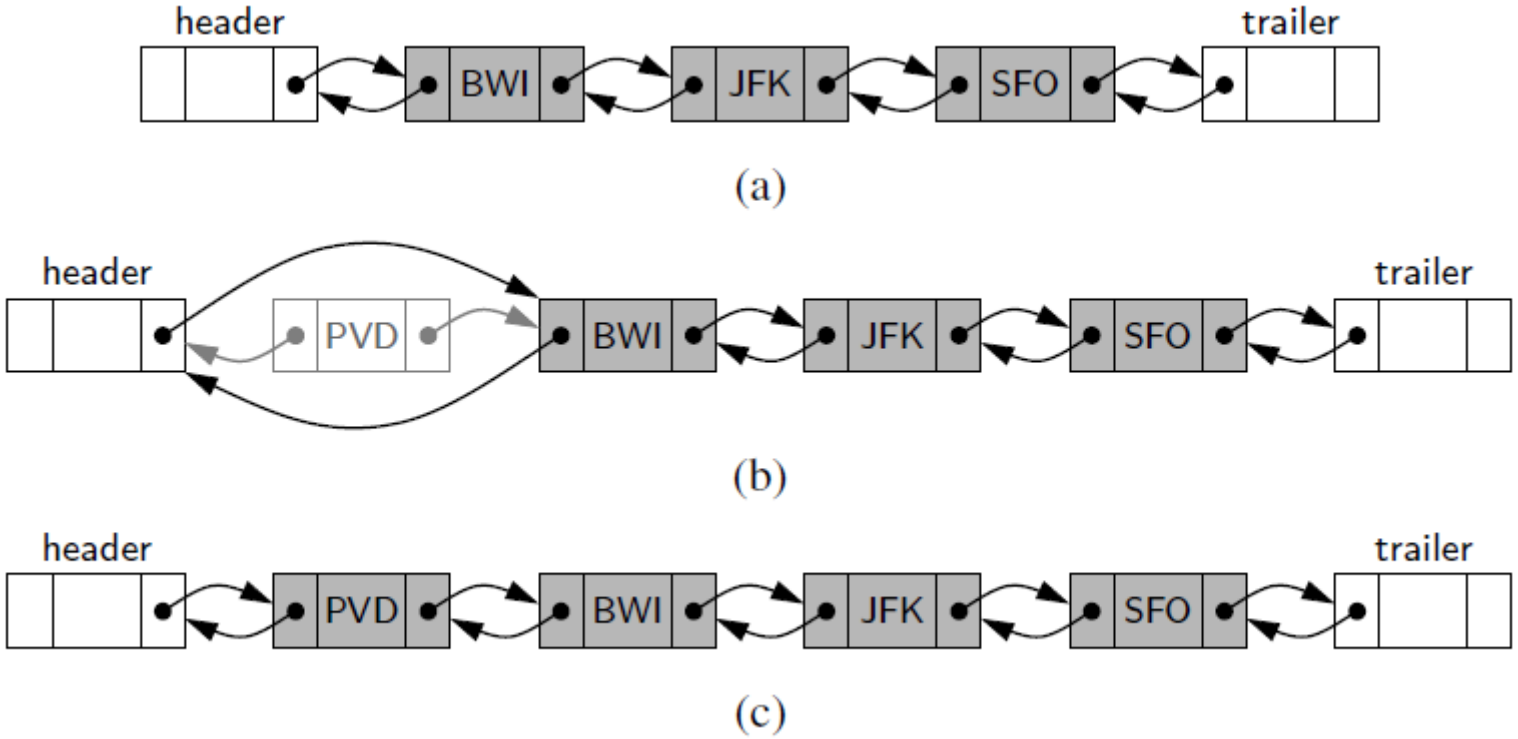
DOUBLY LINKED LISTS
（双链表）

- Inserting into a doubly linked list



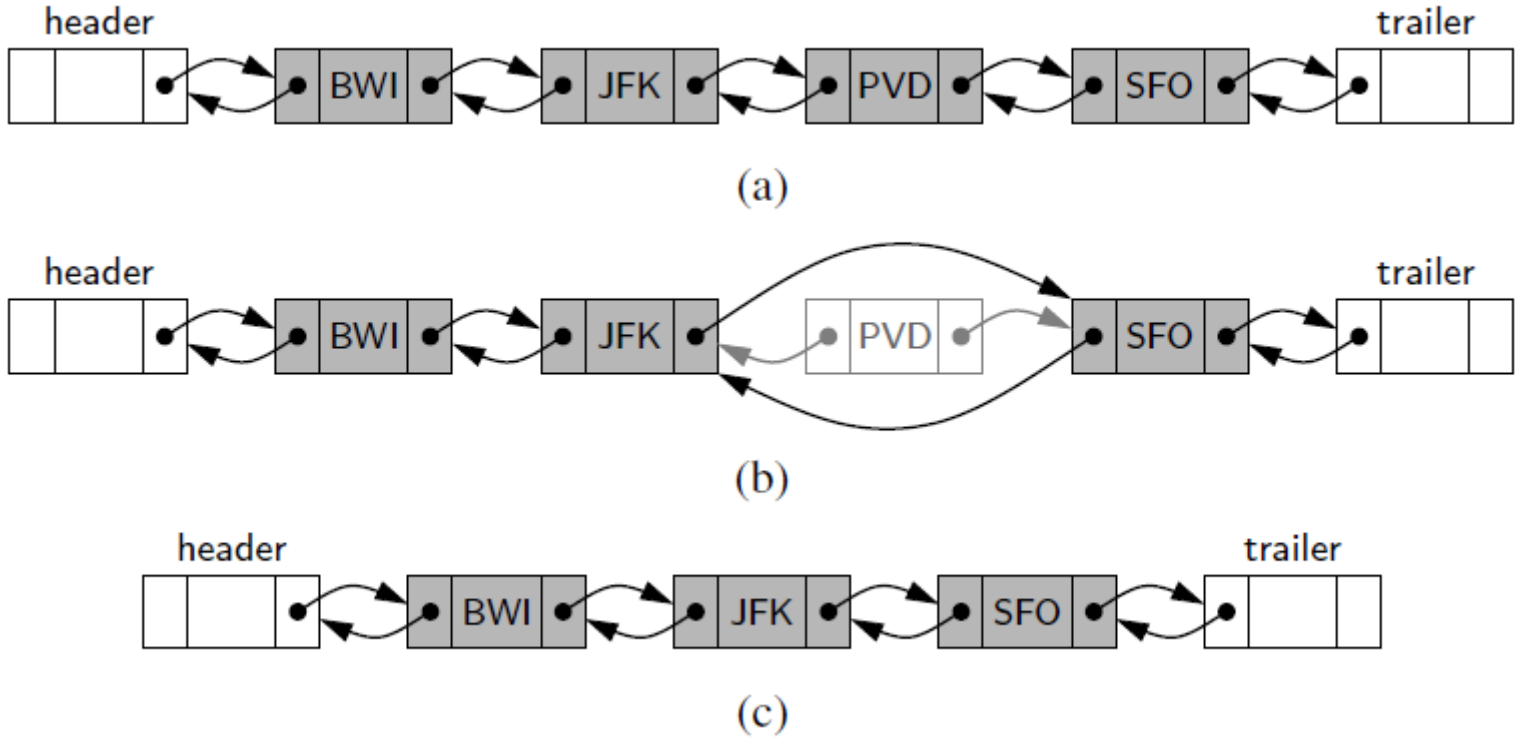
DOUBLY LINKED LISTS
（双链表）

- Inserting into a doubly linked list



DOUBLY LINKED LISTS
（双链表）

- Inserting into a doubly linked list



IMPLEMENTATION OF DOUBLY LINKED LISTS

- Inserting into a doubly linked list

```
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 reference
    self._next = next                               # next node reference
```

IMPLEMENTATION OF DOUBLY LINKED LISTS

- Inserting into a doubly linked list

```
1  class _DoublyLinkedBase:
2    """A base class providing a doubly linked list representation."""
3
4    class _Node:
5      """Lightweight, nonpublic class for storing a doubly linked node."""
6      (omitted here; see previous code fragment)
7
8    def __init__(self):
9      """Create an empty list."""
10     self._header = self._Node(None, None, None)
11     self._trailer = self._Node(None, None, None)
12     self._header._next = self._trailer                # trailer is after header
13     self._trailer._prev = self._header                # header is before trailer
14     self._size = 0                                    # number of elements
15
16   def __len__(self):
17     """Return the number of elements in the list."""
18     return self._size
19
20   def is_empty(self):
21     """Return True if list is empty."""
22     return self._size == 0
```

```
24   def _insert_between(self, e, predecessor, successor):
25     """Add element e between two existing nodes and return new node."""
26     newest = self._Node(e, predecessor, successor)   # linked to neighbors
27     predecessor._next = newest
28     successor._prev = newest
29     self._size += 1
30     return newest
31
32   def _delete_node(self, node):
33     """Delete nonsentinel node from the list and return its element."""
34     predecessor = node._prev
35     successor = node._next
36     predecessor._next = successor
37     successor._prev = predecessor
38     self._size -= 1
39     element = node._element                           # record deleted element
40     node._prev = node._next = node._element = None    # deprecate node
41     return element                                    # return deleted element
```

QUIZ FOR THIS WEEK

- Four people need to cross a rope bridge to get back to their camp
- They only have one flashlight, it only has enough power left to light for 17 minutes
- To cross the bridge, they must use a flashlight
- The bridge is only strong enough to support two people at any time
- Each of the campers walks at a different speed
  - 1 min
  - 2 mins
  - 5 mins
  - 10 mins
- How can the campers make it across in exactly 17 minutes?

THANKS

See you in the next session!