CSI 2103: Data Structures

Linked Lists (Ch 7)

Yonsei University

Spring 2022

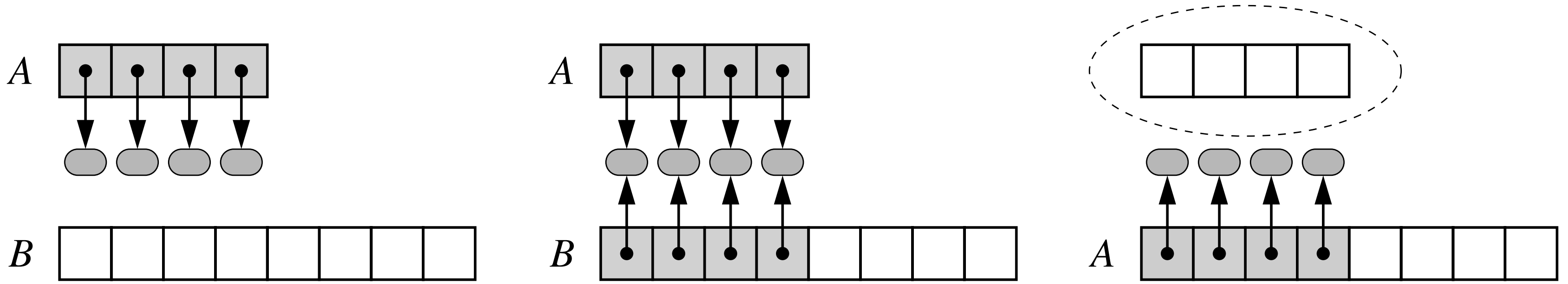
Seong Jae Hwang

Aims

- Linked list: alternative data structure to array
- Pros and cons of linked list
- Singly linked list
- Doubly linked list

Arrays: Drawbacks

- Technically, we need to explicitly do something about the array length
  - What if we need to expand the array length?
  - Dynamic resizing: "Growing" a dynamic array
    1. Create a new array B
    2. Store elements of A in B
    3. Reassign reference A to the new array



Arrays: Drawbacks

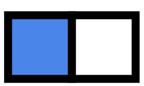
- Each resizing operation is $O(n)$
- So, you may think that appending new elements is going to require many resizing operations, thus appending n elements is $O(n^2)$. This is **not** actually true!
- Amortized running time: the "expected" running time over a long period of time
  - Occasionally doing $O(n)$ is okay!
  - But exactly how occasional?

Quick Detour: Amortized Analysis

- Intuition: An expensive operation is $O(n)$. But if we perform this expensive operation at a rate proportional to n, in the long run, the total cost still grows linearly!
  - For $n = 1000$: after 1000 operations which are $O(1)$, do an expensive operation which is $O(1000)$
  - For $n = 1\mathrm{M}$: after 1M operations which are $O(1)$, do an expensive operation which is $O(1\mathrm{M})$
  - …
  - Does this work when we resize for appending n elements?

Quick Detour: Amortized Analysis

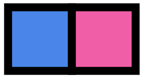
- Goal: Perform a series of n append operations, from an empty array S.
- We pay "cyber-dollars" proportional to n:
  - We "pay" 3 cyber-dollars for each append operation.
  - Each append only "costs" 1 cyber-dollar (cheap)
  - Growing the array from size k to $2k$ "costs" k cyber-dollars (expensive)
  - When we pay more that it costs, we can "save up" for later



Ops	Pays	Costs	Saved for later

Quick Detour: Amortized Analysis

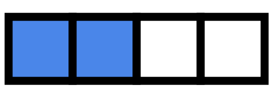
- Goal: Perform a series of n append operations, from an empty array S.
- We pay "cyber-dollars" proportional to n:
  - We "pay" 3 cyber-dollars for each append operation.
  - Each append only "costs" 1 cyber-dollar (cheap)
  - Growing the array from size k to $2k$ "costs" k cyber-dollars (expensive)
  - When we pay more that it costs, we can "save up" for later



Ops	Pays	Costs	Saved for later
Append	$$$	$	$$

Quick Detour: Amortized Analysis

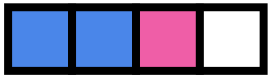
- Goal: Perform a series of n append operations, from an empty array S.
- We pay "cyber-dollars" proportional to n:
  - We "pay" 3 cyber-dollars for each append operation.
  - Each append only "costs" 1 cyber-dollar (cheap)
  - Growing the array from size k to $2k$ "costs" k cyber-dollars (expensive)
  - When we pay more that it costs, we can "save up" for later



Ops	Pays	Costs	Saved for later
Resize		$$	~~$$~~

Quick Detour: Amortized Analysis

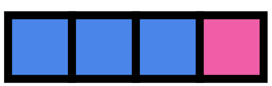
- Goal: Perform a series of n append operations, from an empty array S.
- We pay "cyber-dollars" proportional to n:
  - We "pay" 3 cyber-dollars for each append operation.
  - Each append only "costs" 1 cyber-dollar (cheap)
  - Growing the array from size k to $2k$ "costs" k cyber-dollars (expensive)
  - When we pay more that it costs, we can "save up" for later



Ops	Pays	Costs	Saved for later
Append	$$$	$	$$

Quick Detour: Amortized Analysis

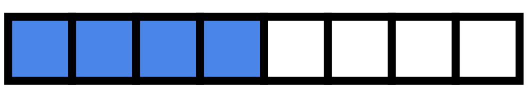
- Goal: Perform a series of n append operations, from an empty array S.
- We pay "cyber-dollars" proportional to n:
  - We "pay" 3 cyber-dollars for each append operation.
  - Each append only "costs" 1 cyber-dollar (cheap)
  - Growing the array from size k to $2k$ "costs" k cyber-dollars (expensive)
  - When we pay more that it costs, we can "save up" for later



Ops	Pays	Costs	Saved for later
Append	\$\$\$	\$	\$\$\$\$

Quick Detour: Amortized Analysis

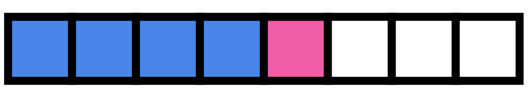
- Goal: Perform a series of n append operations, from an empty array S.
- We pay "cyber-dollars" proportional to n:
  - We "pay" 3 cyber-dollars for each append operation.
  - Each append only "costs" 1 cyber-dollar (cheap)
  - Growing the array from size k to $2k$ "costs" k cyber-dollars (expensive)
  - When we pay more that it costs, we can "save up" for later



Ops	Pays	Costs	Saved for later
Resize		$$$$	~~$$$$~~

Quick Detour: Amortized Analysis

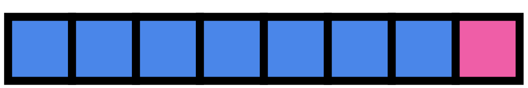
- Goal: Perform a series of n append operations, from an empty array S.
- We pay "cyber-dollars" proportional to n:
  - We "pay" 3 cyber-dollars for each append operation.
  - Each append only "costs" 1 cyber-dollar (cheap)
  - Growing the array from size k to $2k$ "costs" k cyber-dollars (expensive)
  - When we pay more that it costs, we can "save up" for later



Ops	Pays	Costs	Saved for later
Append	$$$	$	$$

Quick Detour: Amortized Analysis

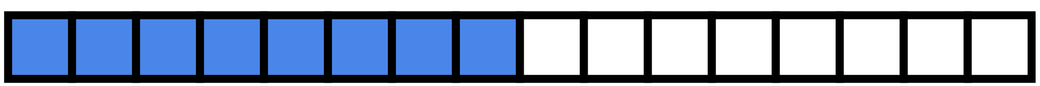
- Goal: Perform a series of n append operations, from an empty array S.
- We pay "cyber-dollars" proportional to n:
  - We "pay" 3 cyber-dollars for each append operation.
  - Each append only "costs" 1 cyber-dollar (cheap)
  - Growing the array from size k to $2k$ "costs" k cyber-dollars (expensive)
  - When we pay more that it costs, we can "save up" for later



Ops	Pays	Costs	Saved for later
Append	\$\$\$	\$	\$\$\$\$\$\$\$\$

Quick Detour: Amortized Analysis

- Goal: Perform a series of n append operations, from an empty array S.
- We pay "cyber-dollars" proportional to n:
  - We "pay" 3 cyber-dollars for each append operation.
  - Each append only "costs" 1 cyber-dollar (cheap)
  - Growing the array from size k to $2k$ "costs" k cyber-dollars (expensive)
  - When we pay more that it costs, we can "save up" for later
- We always "pay" 3 cyber-dollars per append: $O(3n) = O(n)$



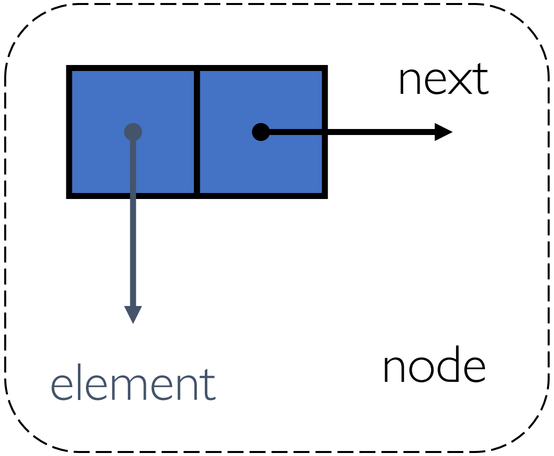
Ops	Pays	Costs	Saved for later
Resize		$$$$$$$$	~~$$$$$$$$~~

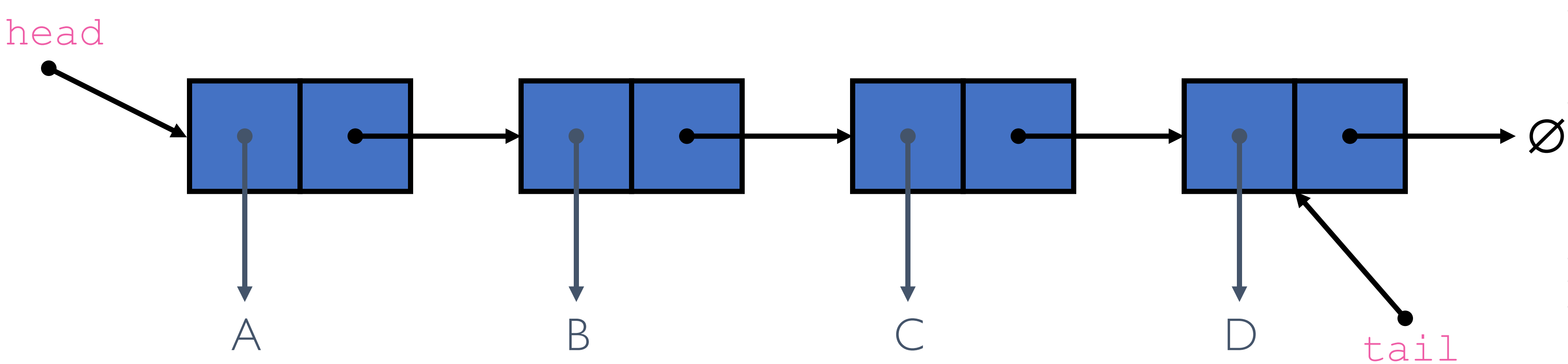
Arrays: Drawbacks

- Adding at earlier indices (i.e., index 0) may require unnecessarily many operations
  - Adding at 0 index requires the shifting of all entries
- Usually implemented with a contiguous block in memory
  - Concatenation of two arrays requires a new initialization?
  - If the pre-existing objects are not in contiguous blocks?
- Can we add a little more flexibility?

Singly Linked List

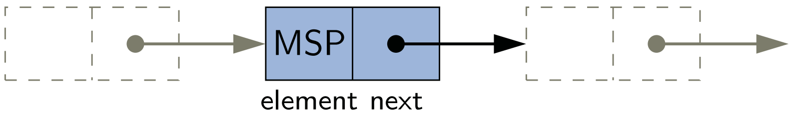
- Sequence of nodes
  - Element
  - Link to the next node
- `head` pointer to the first node of list
- `tail` pointer to the last node of list
- Explicitly connecting entries





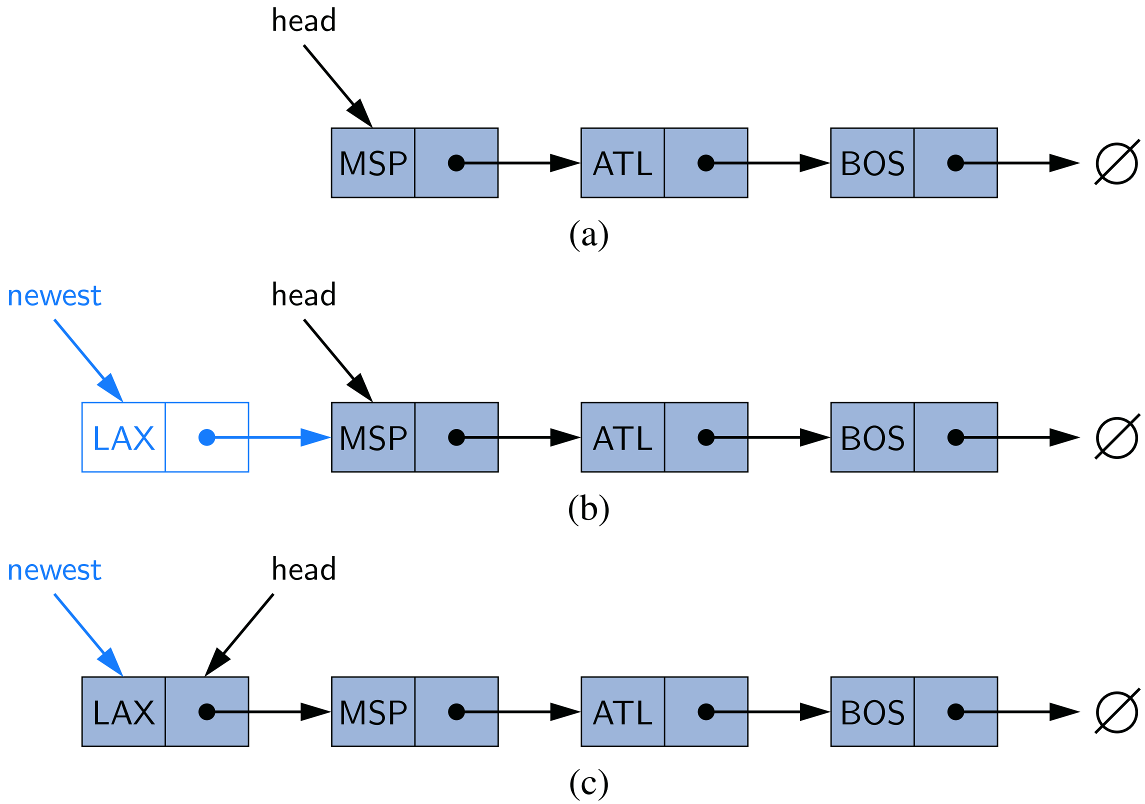
Example

- Node of airport code



Example: Inserting at the head

- Insert a new node with LAX element at the head



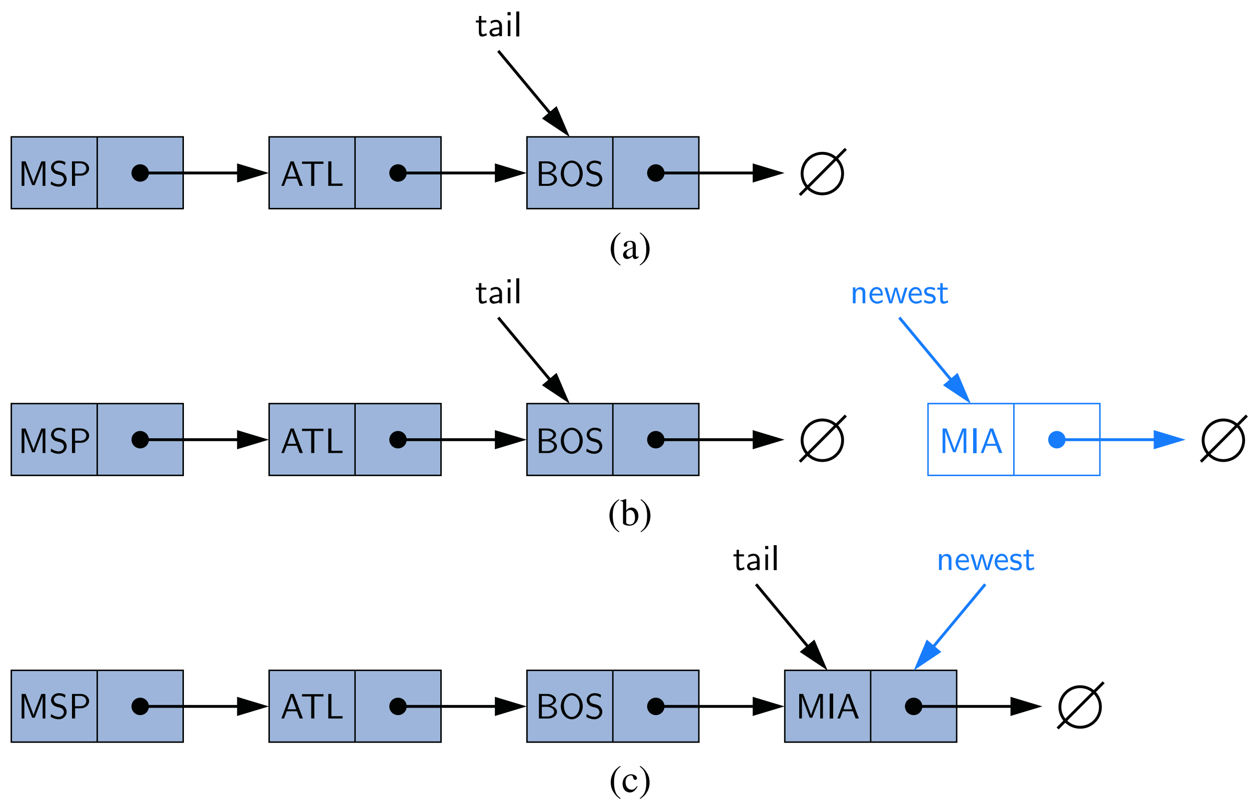
Move head after new node is connected!

Algorithm addFirst(e):

newest = Node(e) {create new node instance storing reference to element e}
newest.next = head {set new node's next to reference the old head node}
head = newest {set variable head to reference the new node}
size = size + 1 {increment the node count}

Example: Inserting at the tail

- Insert a new node with MIA element at the tail



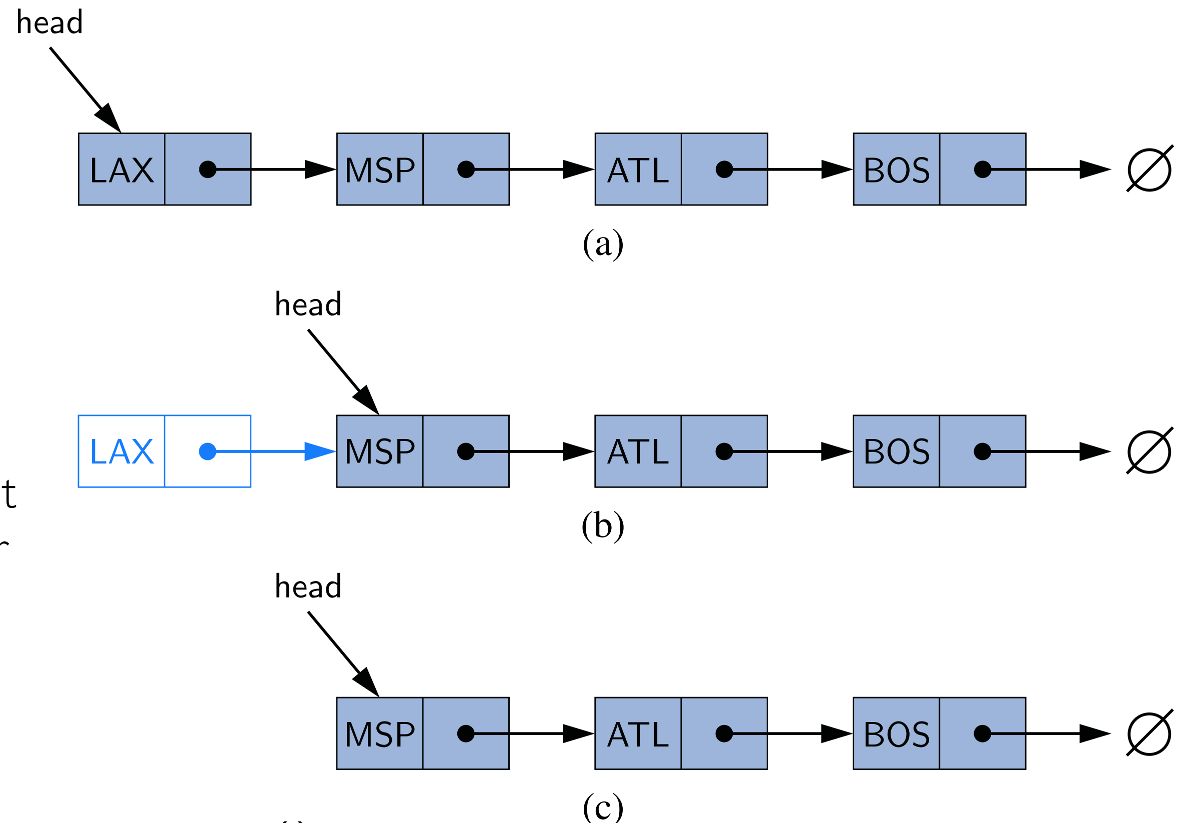
Move tail after new node is connected!

Algorithm addLast(e):

newest = Node(e) {create new node instance storing reference to element e}
newest.next = null {set new node's next to reference the null object}
tail.next = newest {make old tail node point to new node}
tail = newest {set variable tail to reference the new node}
size = size + 1 {increment the node count}

Example: Removing at the head

- Remove a node at the head



Disconnect node after head has moved

Algorithm removeFirst( ):

if head == null **then**

the list is empty.

head = head.next {make head point to next node (or null)}

size = size − 1 {decrement the node count}

Implementation

- Basic functions of a SinglyLinkedList class
- We will be generic about the element types

size( ):	Returns the number of elements in the list.
isEmpty( ):	Returns **true** if the list is empty, and **false** otherwise.
first( ):	Returns (but does not remove) the first element in the list.
last( ):	Returns (but does not remove) the last element in the list.
addFirst(e):	Adds a new element to the front of the list.
addLast(e):	Adds a new element to the end of the list.
removeFirst( ):	Removes and returns the first element of the list.

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
```

SinglyLinkedList class

```
def __init__(self):
    """Create an empty SLL."""
    self._head = None
    self._tail = None
    self._size = 0

def __len__(self):
    """Return the number of elements in the SLL."""
    return self._size

def isEmpty(self):
    """Return True if the SLL is empty."""
    return self._size == 0
```

SinglyLinkedList class

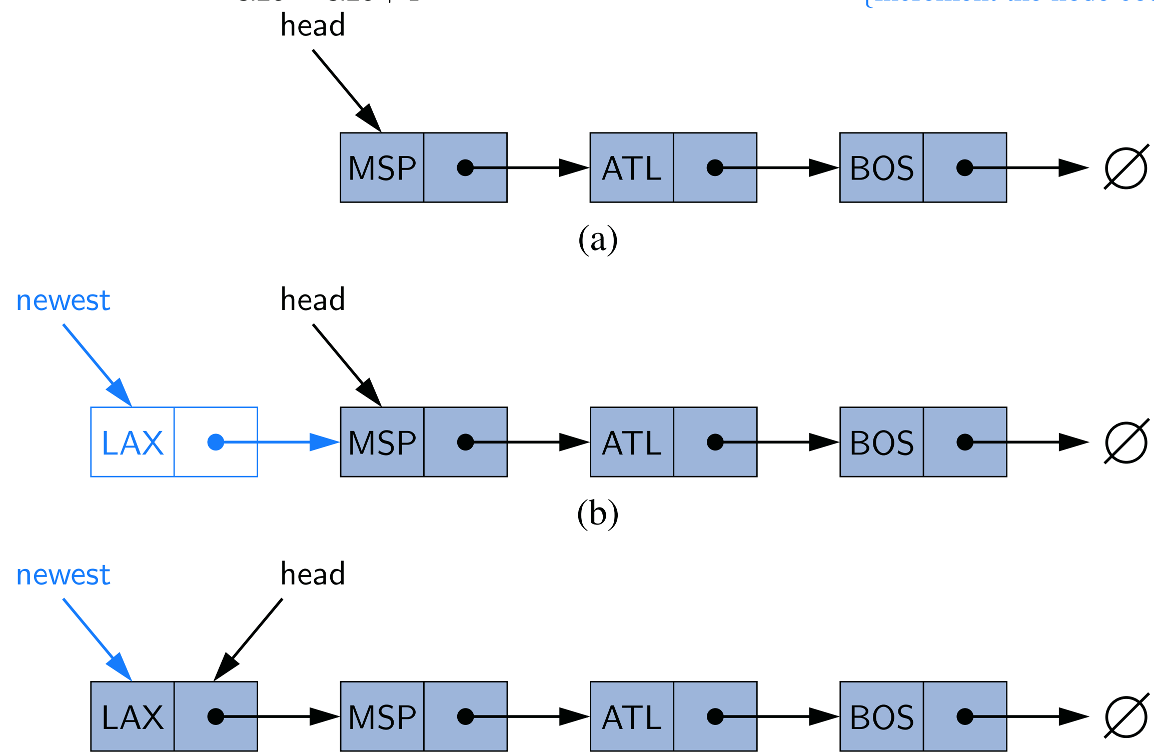
```
def first(self):
  """Return (but do not remove) the first element."""
  if self.isEmpty():
    raise Empty('Stack is empty')    # Exception we define later
  return self._head._element

def last(self):
  """Return (but do not remove) the last element."""
  if self.isEmpty():
    raise Empty('Stack is empty')    # Exception we define later
  return self._tail._element
```

SinglyLinkedList class

```python
def addFirst(self, e):
    """Add element e to the front of SLL."""
    self._head = self._Node(e, self._head)
    self._size += 1
```

Algorithm addFirst(e):
  newest = Node(e) {create new node instance storing reference to element e}
  newest.next = head {set new node's next to reference the old head node}
  head = newest {set variable head to reference the new node}
  size = size + 1 {increment the node count}

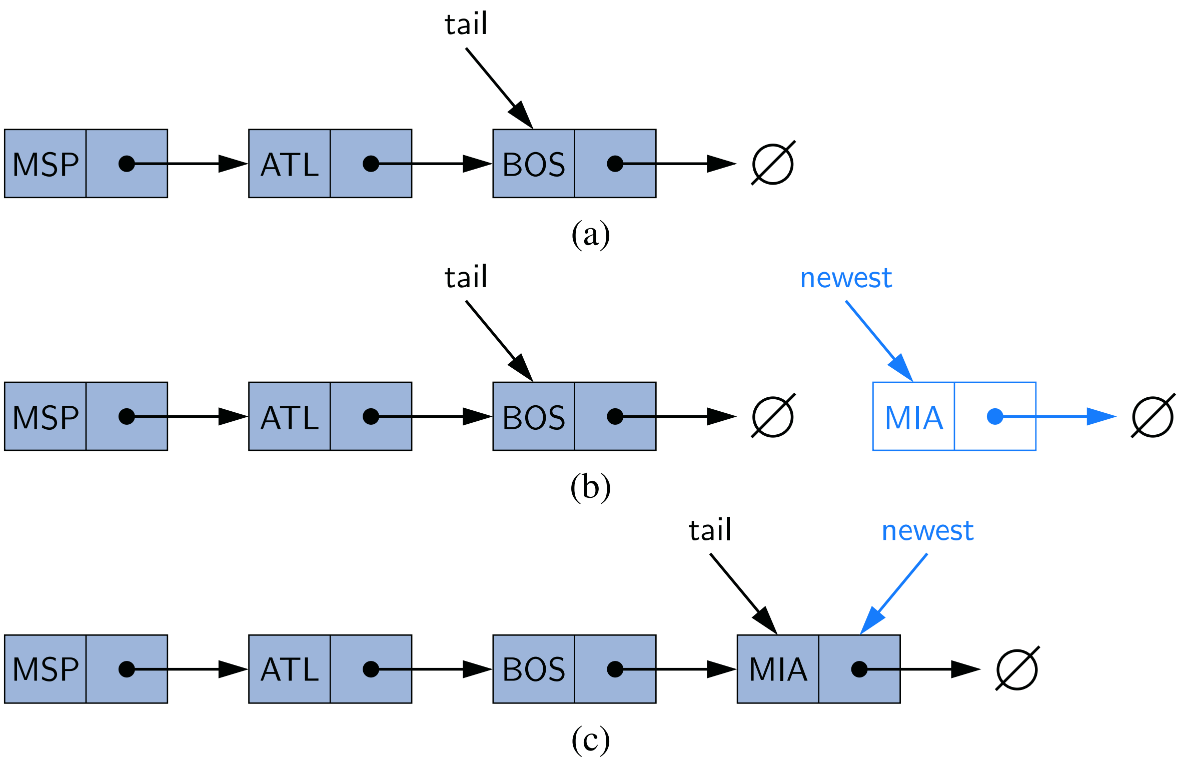


(c)

SinglyLinkedList class

```
def addLast(self, e):
  """Add element e to the end of SLL."""
  newest = self._Node(e, None)
  if self.isEmpty():
    self._head = newest
  else:
    self._tail._next = newest
  self._tail = newest
  self._size += 1
```

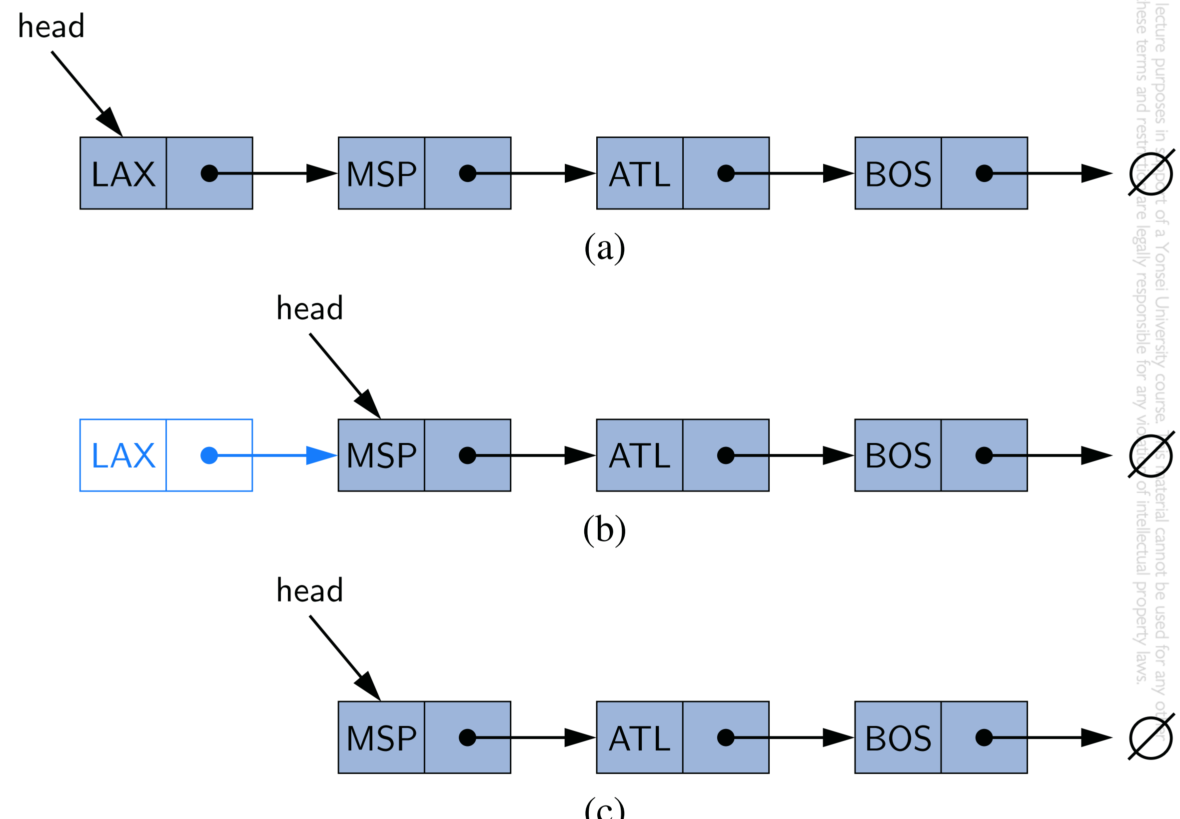
Algorithm addLast(e):
- newest = Node(e) {create new node instance storing reference to element e}
- newest.next = null {set new node's next to reference the null object}
- tail.next = newest {make old tail node point to new node}
- tail = newest {set variable tail to reference the new node}
- size = size + 1 {increment the node count}



SinglyLinkedList class

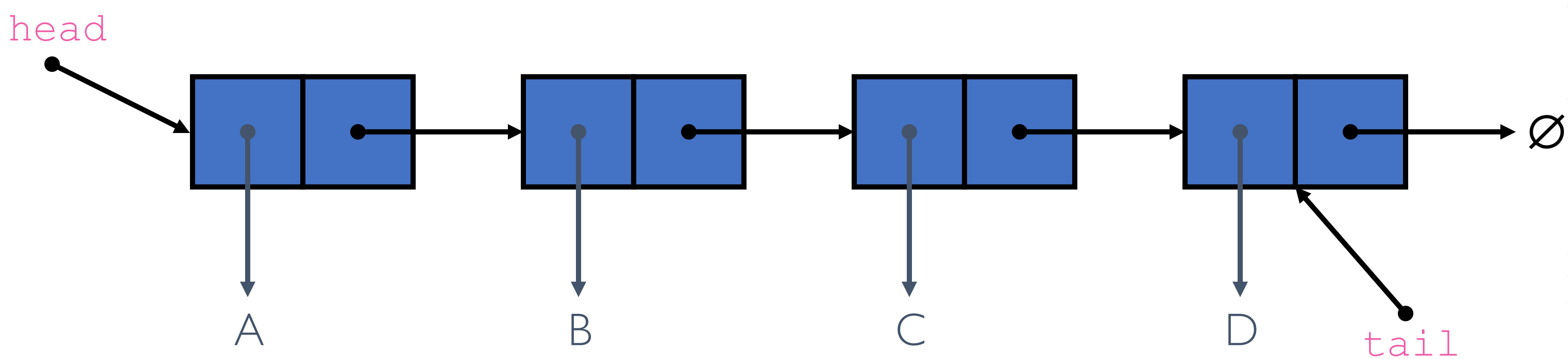
```python
def removeFirst(self):
  """Remove and return the first element of the SLL."""
  if self.isEmpty():
    raise Empty('Queue is empty')
  answer = self._head._element
  self._head = self._head._next
  self._size -= 1
  if self.isEmpty():
    self._tail = None
  return answer
```

Algorithm removeFirst():
  if head == null **then**
    the list is empty.
  head = head.next {make head point to next node (or null)}
  size = size − 1 {decrement the node count}



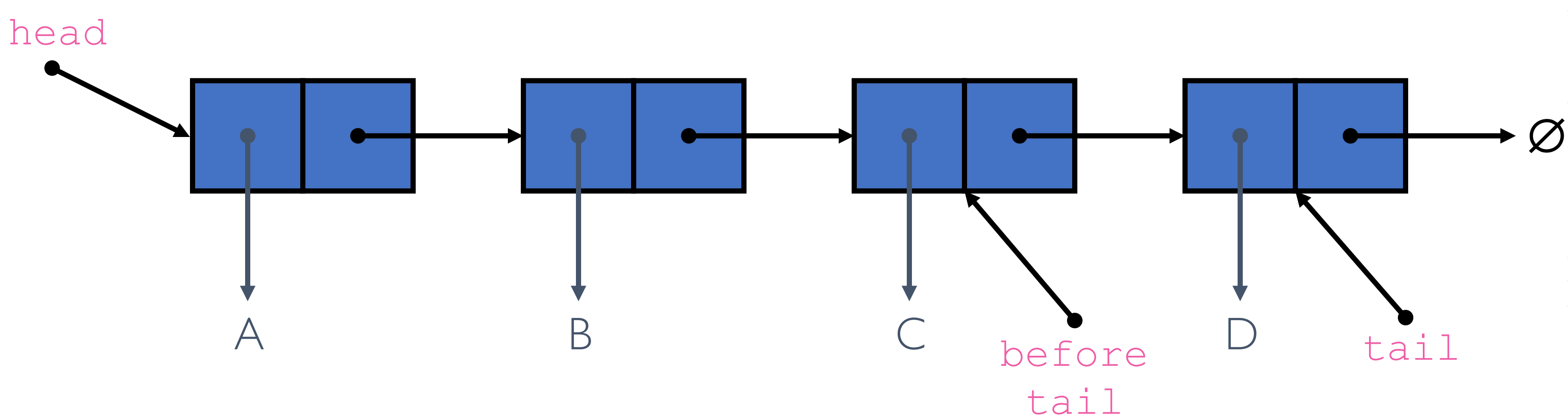
Removing at the tail?

- Remove a node at the tail
- What's the issue?



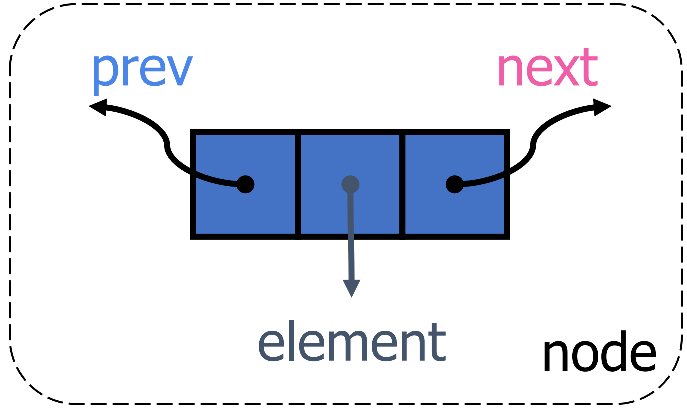
Removing at the tail

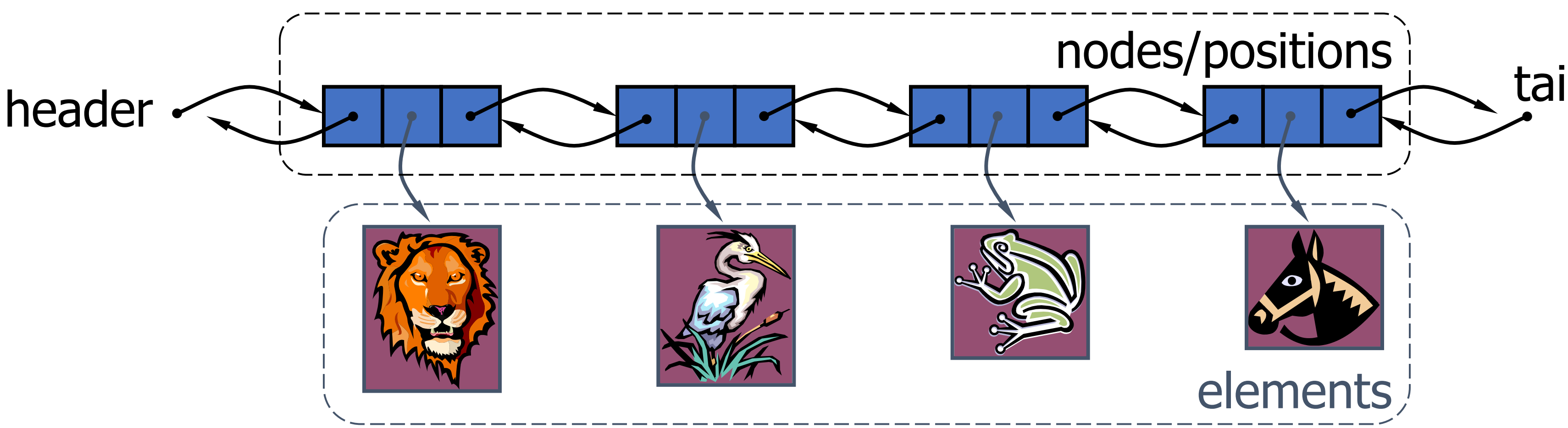
- Remove a node at the tail
- Need a pointer to the node just before tail
- Since the list can only be traversed forward via next, we cannot directly reach the node before tail
- Can we somehow traverse backward via…?



Doubly Linked List

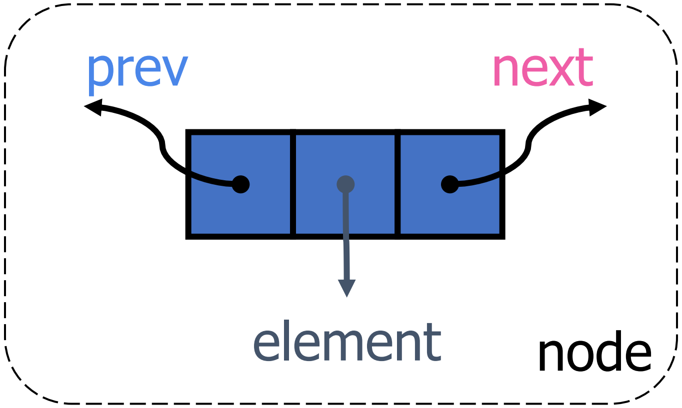
- Sequence of nodes
  - Element
  - Link to the next node
  - Link to the previous node

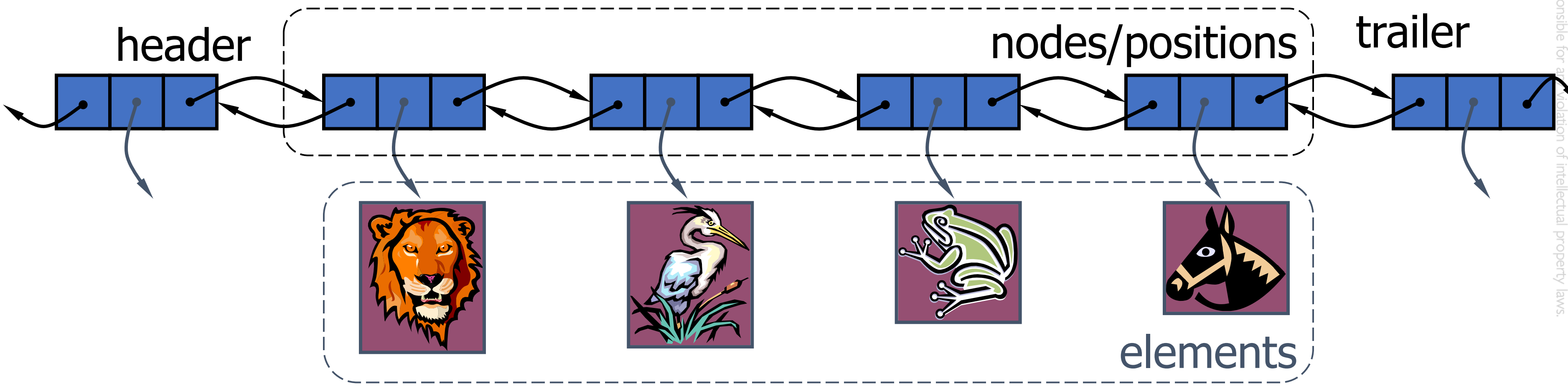




Doubly Linked List

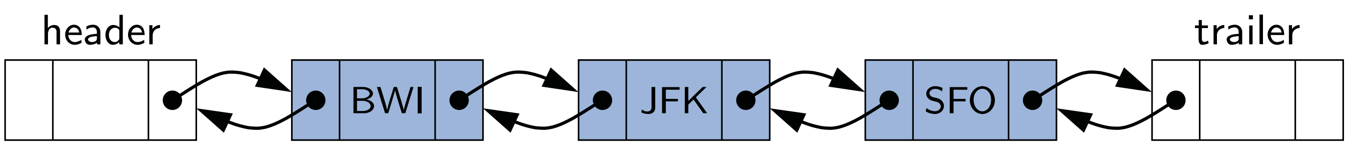
- Sequence of nodes
  - Element
  - Link to the next node
  - Link to the previous node
- variant: header node and trailer node
  - Treating them as “nodes” generalizes many operations



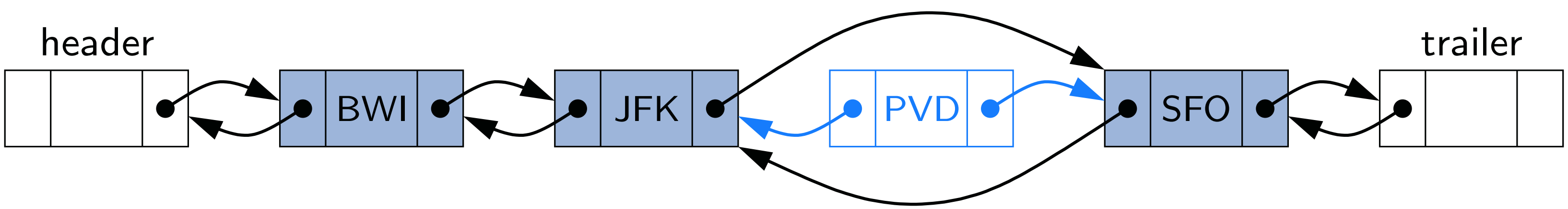


Example: Inserting

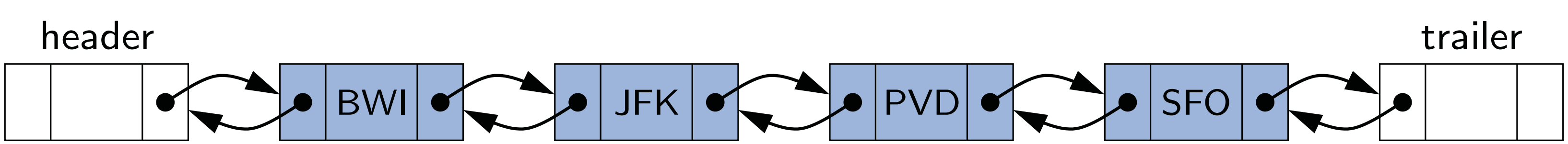
- Since header and trailer are nodes, every insertion follows the same operation with no head/tail corner cases
  - i.e., PVD can be inserted at the front (before BWI) and end (after SFO) and still expect the same operation



(a)



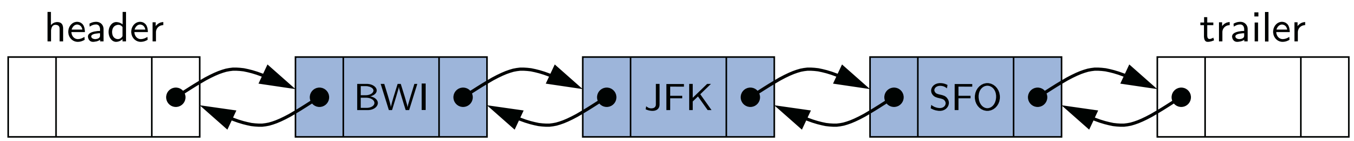
(b)



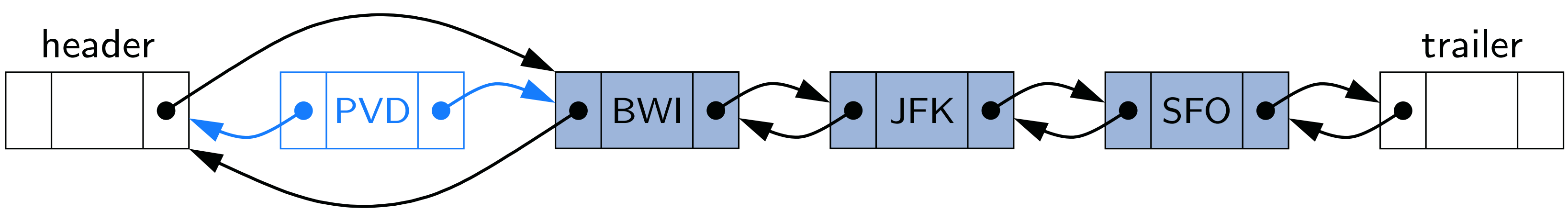
(c)

Example: Inserting

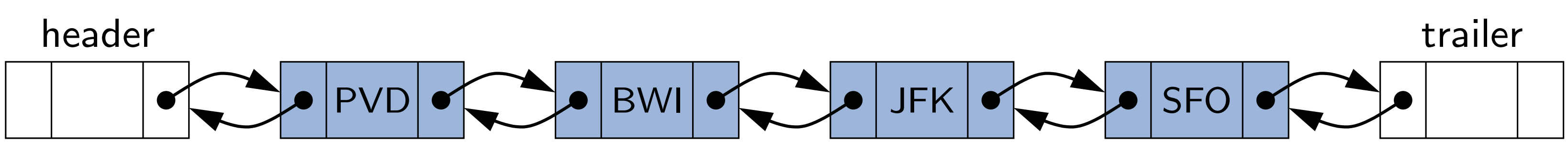
- Since header and trailer are nodes, every insertion follows the same operation with no head/tail corner cases
  - i.e., PVD can be inserted at the front (before BWI) and end (after SFO) and still expect the same operation



(a)



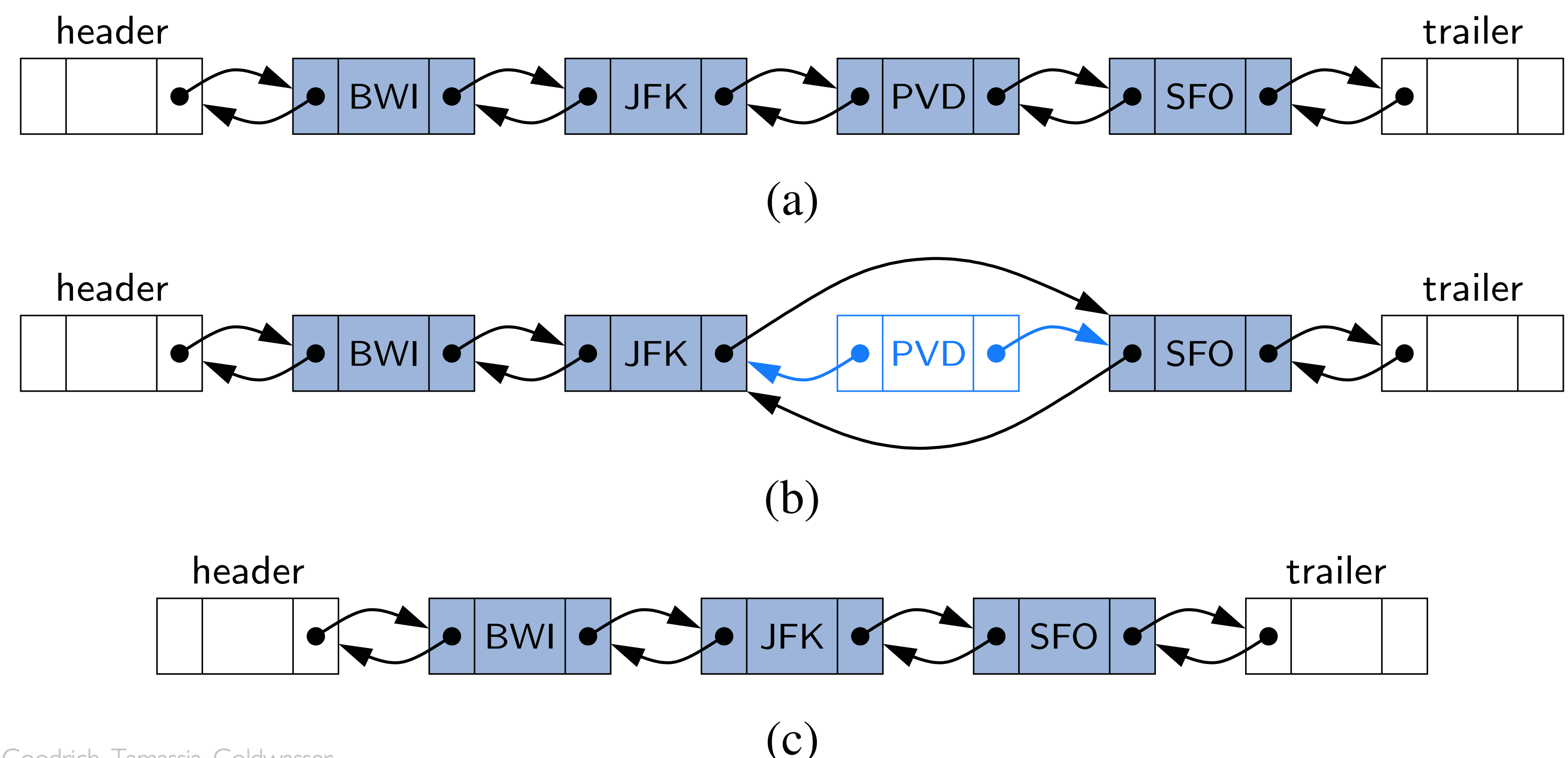
(b)



(c)

Example: Deleting

- Deletion just needs to remove the references pointing to the entry you want to delete
    - Nothing is pointing at PVD, so it will be reclaimed by the system



Implementation

- Basic functions of a DoublyLinkedList class
- Very similar functions, except we now have removeLast!

size( ): Returns the number of elements in the list.

isEmpty( ): Returns **true** if the list is empty, and **false** otherwise.

first( ): Returns (but does not remove) the first element in the list.

last( ): Returns (but does not remove) the last element in the list.

addFirst(e): Adds a new element to the front of the list.

addLast(e): Adds a new element to the end of the list.

removeFirst( ): Removes and returns the first element of the list.

removeLast( ): Removes and returns the last element of the list.

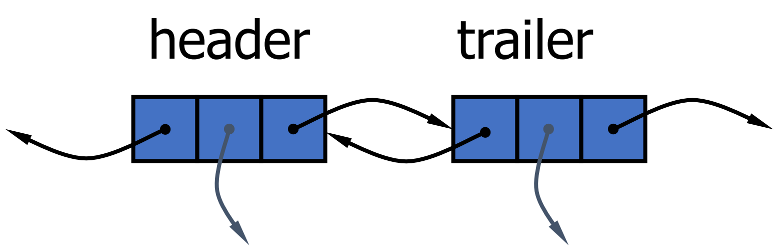
DoublyLinkedList class

```
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
```

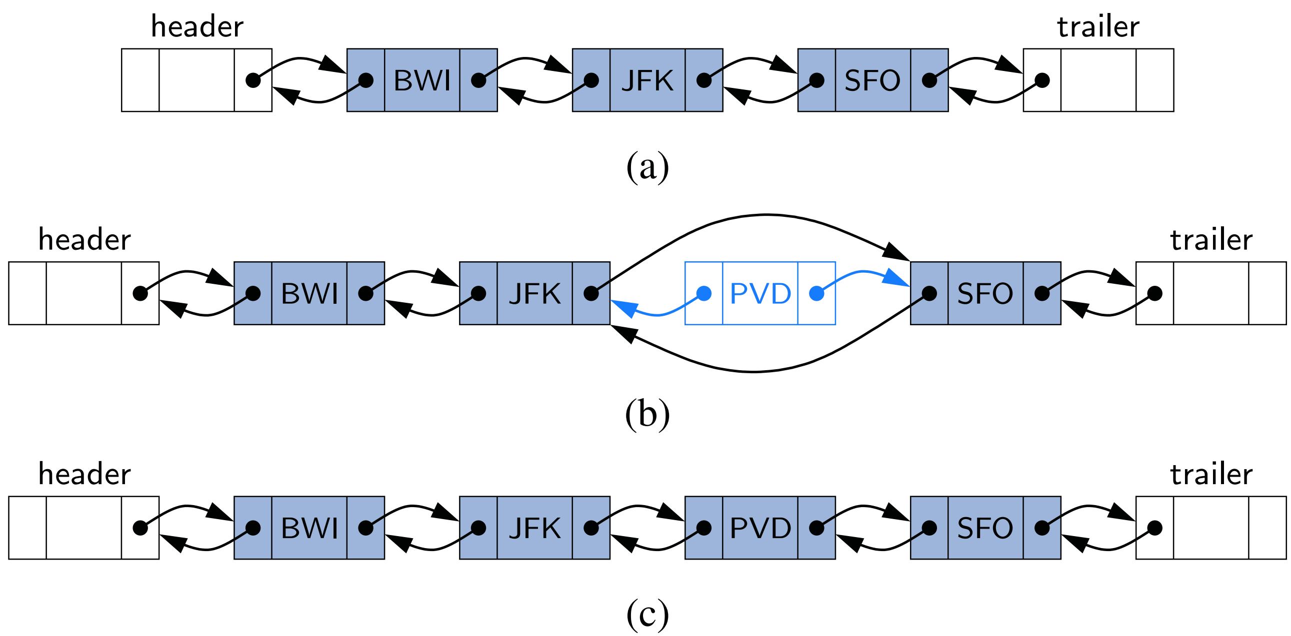
DoublyLinkedList class

```
def __init__(self):
    """Create an empty DLL."""
    self._header = self._Node(None, None, None)
    self._trailer = self._Node(None, None, None)
    self._header._next = self.trailer
    self._trailer._prev = self._header
    self._size = 0
```



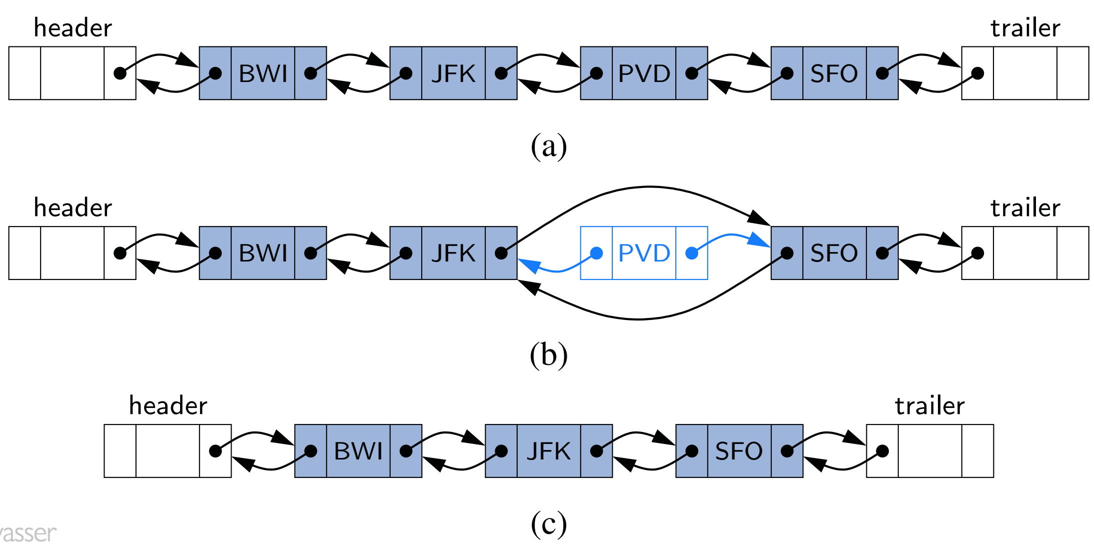
DoublyLinkedList class

```
def _insert_between(self, e, predecessor, successor):
    """Add element e between two existing nodes."""
    newest = self._Node(e, predecessor, successor)
    predecessor._next = newest
    successor._prev = newest
    self._size += 1
```



DoublyLinkedList class

```
def _delete_node(self, node):
  """Delete node from the list."""
  predecessor = node._prev
  successor = node._next
  predecessor._next = successor
  successor._prev = predecessor
  self._size = -= 1
```



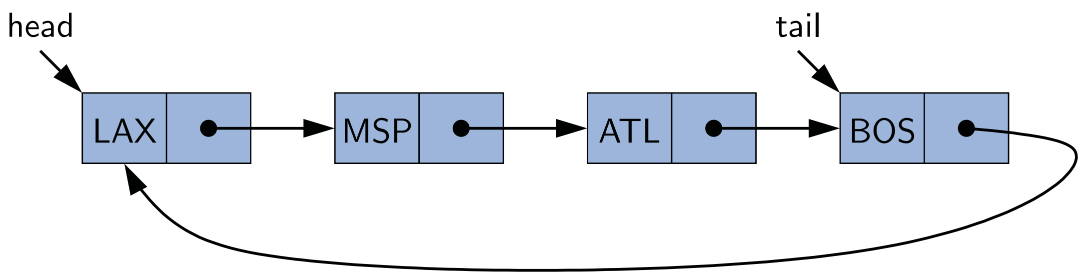
Time Complexity

- More operations
  - Insert/Delete: assuming the location has been found
  - InsertAt/DeleteAt: Access + Insert/Delete

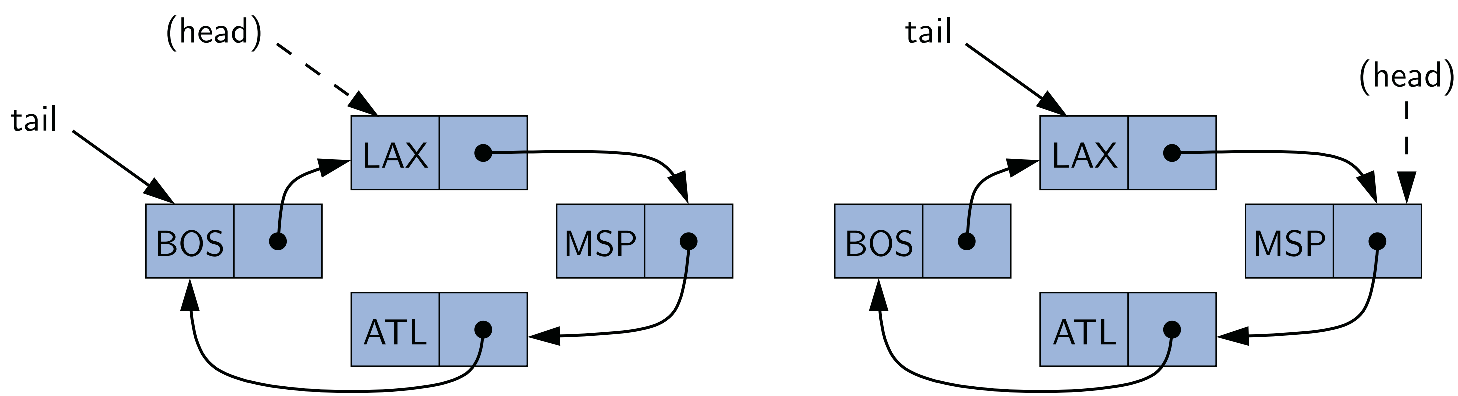
Operation	Description	Unsorted Array	Singly Linked List	Doubly Linked List
Access(i)	Accessing entry at i'th location/index	$O(1)$	$O(n)$	$O(n)$
SearchFor(e)	Searching for specific entry e	$O(n)$	$O(n)$	$O(n)$
Insert(e)	Insert entry e (location found already)	$O(n)$	$O(1)$	$O(1)$
Delete(e)	Delete entry e (location found already)	$O(n)$	$O(1)$	$O(1)$
InsertAt(i,e)	Insert entry e at location i	$O(n)$	$O(n)$	$O(n)$
DeleteAt(i,e)	Delete entry e at location i	$O(n)$	$O(n)$	$O(n)$
InsertAtFirst(e)	Insert entry e at first	$O(n)$	$O(1)$	$O(1)$
InsertAtLast(e)	Insert entry e at last	$O(1)$	$O(n)$	$O(1)$

Extra: Circularly Linked List

- Useful for round-robin operations



- Rotate operation: move the first element to the end of the list
  - Circulate through the elements in the list
  - (Won't talk about this much)



(a) (b)

Summary

- Singly Linked List vs. Doubly Linked List
  - Less rigid, more flexible
- Time complexity analysis
  - Again, trade-offs

- Next:
  - Note that some operations require repetitions
  - Before we move on to other data structures, we will quickly discuss a technique for repetitive, recursive tasks
  - This technique will come up often in data structures