Data Structures in Python

8. Linked Lists

Prof. Moheb Ramzy Girgis
Department of Computer Science
Faculty of Science
Minia University

Introduction

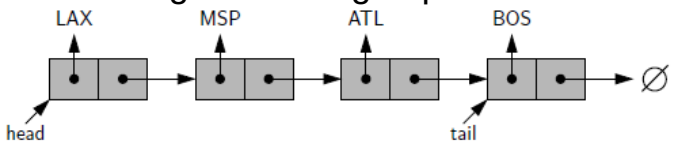
- Python's list class is highly optimized, and often a great choice for storage.
- But, there are some notable disadvantages:
  - The length of a dynamic array might be longer than the actual number of elements that it stores.
  - Insertions and deletions at interior positions of an array are expensive.
- In this lecture, we introduce a data structure known as a ***linked list***, which provides an alternative to an array-based sequence (such as a Python list).
- Like array-based sequences, linked lists keep elements in a certain order.
- In a linked list a lightweight object, known as a ***node***, is allocated for each element.

Introduction

- Each node maintains a reference to its element and one or more references to neighboring nodes in order to collectively represent the linear order of the sequence.
- We will demonstrate a trade-off of advantages and disadvantages when contrasting array-based sequences and linked lists.
- Elements of a linked list cannot be efficiently accessed by a numeric index *k*, and we cannot tell just by examining a node if it is the second, fifth, or twentieth node in the list.
- However, linked lists avoid the two disadvantages noted above for array-based sequences.

Singly Linked Lists

- 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,
  - a reference to the ***next node*** of the list.
- The figure shows an example of a singly linked list whose elements are strings indicating airport codes.



- The *first* and *last* node of a linked list are known as the ***head*** and ***tail*** of the list, respectively.
- We can identify the ***tail*** as the node having ***None*** as its next reference. The ***None object*** is denoted as ∅

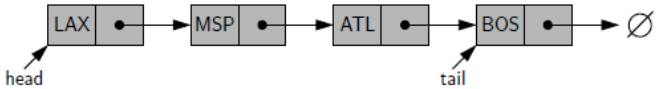
Singly Linked Lists

- 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.
- 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***.
- A linked list's representation in memory relies on the collaboration of many objects:
  - Each node is represented as a unique object, with that instance storing a reference to its element and a reference to the next node (or ***None***).
  - Another object represents the linked list as a whole. This ***linked list instance*** must keep a reference to the ***head*** of the list.

5

Singly Linked Lists

- There is no need to store a direct reference to the ***tail*** of the list, as it could be located by starting at the ***head*** and traversing the rest of the list.
- However, storing an explicit reference to the ***tail*** node is a common convenience to avoid such a traversal.
- It is also common for the linked list instance to keep a count of the total number of nodes that comprise the list (known as the ***size*** of the list), to avoid the need to traverse the list to count the nodes.
- For simplicity, we illustrate a node's element embedded directly within the node structure, as shown in the figure, even though the element is, in fact, an independent object.



Data Structures in Python - Prof. Moheb Ramzy Girgis Dept. of Computer Science - Faculty of Science Minia University

6

Singly Linked Lists

➢ ***Traversing a Singly Linked List***

- Sometimes we need to sequentially traverse the elements of a linked list and process them, e.g. print them.
- To traverse a list, we start at the ***head***, and moving from one node to another by following each node's next reference, until we reach the ***tail*** of the list.
- We use an identifier ***current*** to represent the current node.
- Initially, we set ***current*** to reference the ***head*** node, then by setting ***current = current.next***, we can effectively advance through the nodes of the list.
- The pseudo-code for traversing a list:

```
Algorithm traverse(L):
  current = L.head     # set current to reference the head node
  while (current) {
      e = current.element   # get the element at the current node
      process(e)            # process the current element
      current = current.next          # move to the next node
  }
```

7

Singly Linked Lists

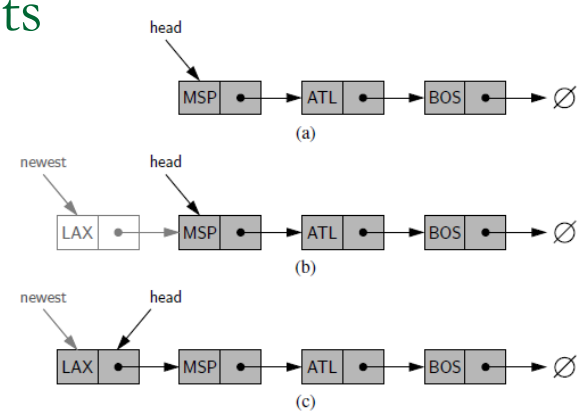
➢ ***Inserting an Element at the Head of a Singly Linked List***

- An important property of a linked list is that it does not have a predetermined fixed size; it uses space proportionally to its current number of elements.
- We can easily insert an element at the head of the list:
  - create a new node,
  - set its element to the new element,
  - set its next link to refer to the current head,
  - set the list's head to point to the new node.
- The corresponding pseudo-code:

```
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}
```

Singly Linked Lists

- The figure illustrates the insertion of an element at the head of a singly linked list: (a) before the insertion; (b) after creation of a new node; (c) after reassignment of the head reference.



  - In the above pseudo-code, note that we set the next pointer of the new node ***before*** we reassign variable L.head to it.
  - If the list were initially empty (i.e., L.head is ***None***), then a natural consequence is that the new node has its next reference set to ***None***.

Singly Linked Lists

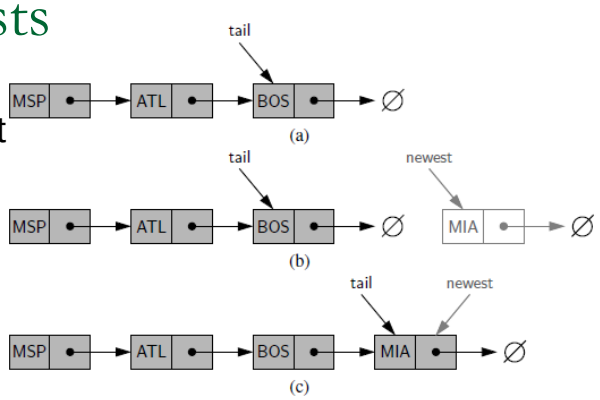
➢ ***Inserting an Element at the Tail of a Singly Linked List***

- We can also easily insert an element at the tail of the list, provided we keep a reference to the tail node:
  - create a new node,
  - assign its next reference to ***None***,
  - set the next reference of the tail to point to this new node,
  - update the tail reference itself to this new node.
- The corresponding pseudo-code:

```
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}
```

Singly Linked Lists

- The figure illustrates the insertion of an element at the ***tail*** of a singly linked list: (a) before the insertion; (b) after creation of a new node; (c) after reassignment of the tail reference.



  - In the above pseudo-code, note that we set the next pointer for the old tail node ***before*** we make variable tail point to the new node.
  - This code would need to be adjusted for inserting onto an empty list, since there would not be an existing tail node.

Singly Linked Lists

➢ ***Removing an Element from the Head of a Singly Linked List***

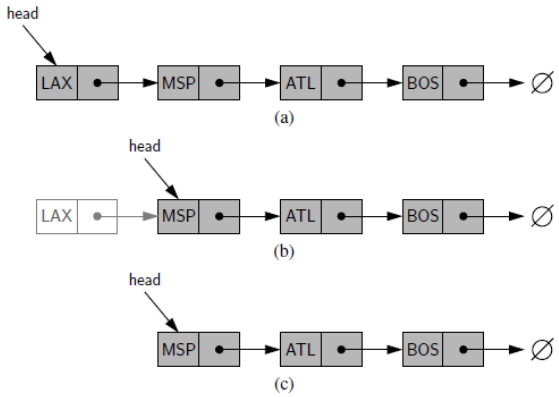
- Removing an element from the ***head*** of a singly linked list is essentially the reverse operation of inserting a new element at the head.
- The corresponding pseudo-code:

```
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}
```

- Unfortunately, we cannot easily delete the last node of a singly linked list.
- Even if we maintain a tail reference directly to the last node of the list, we must be able to access the node ***before*** the last node in order to remove the last node.

Singly Linked Lists

- But we cannot reach the node before the ***tail*** by following next links from the ***tail***.
- The only way to access this node is to start from the ***head*** of the list and search all the way through the list.
- But such a sequence of link-hopping operations could take a long time.
- To support such an operation efficiently, we need to make our list ***doubly linked***, as we will see later.
- The figure illustrates the removal of an element at the head of a singly linked list: (a) before the removal; (b) after "linking out" the old head; (c) final configuration.



Implementing a Stack with a Singly Linked List

- Now, we will see how to implement the ***stack ADT*** in Python with a ***Singly Linked List***.
- Since all ***stack*** operations affect the top, and we can efficiently insert and delete elements in constant time only at the head, we orient the ***top*** of the stack at the ***head*** of our list.
- To represent individual nodes of the list, we develop a lightweight **_Node** class.
- This class will never be directly exposed to the user of our stack class, so we will formally define it as a nonpublic, nested class of our ***LinkedStack class***.
- A node has only two instance variables: _element and _next.
- We intentionally define **__slots__** in the **_Node** class to provide greater efficiency in memory usage, because there may be many node instances in a single list.

14

Implementing a Stack with a Singly Linked List

- The constructor of the **_Node** class is designed to specify initial values for both fields of a newly created node.
- The definition of the **_Node** class is shown below.

```
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
```

- In our ***LinkedStack*** class, each stack instance maintains two variables:
  - the ***_head*** member, which is a reference to the node at the head of the list (or None, if the stack is empty).
  - the ***_size*** instance variable that keeps track of the current number of elements in the stack.

Implementing a Stack with a Singly Linked List

- The implementation of ***push*** essentially mirrors the pseudo-code for ***insertion at the head*** of a singly linked list.
- To push a new element ***e*** onto the stack, we do the following changes to the linked structure:
  - Invoke the ***constructor*** of the ***_Node*** class as follows:

```
self._head = self._Node(e, self._head) # create and link a new node
```

    which sets the ***_next*** field of the new node to the ***existing*** top node, and reassigns ***self._head*** to the new node.
- The ***top*** method should return the ***element*** that is at the top of the stack.
  - When the stack is empty, we raise an ***Empty exception***, as defined in Lecture 6.
  - When the stack is nonempty, ***self._head*** is a reference to the ***first node*** of the linked list. So, the ***top*** element can be identified as ***self._head._element***.

Implementing a Stack with a Singly Linked List

- The implementation of ***pop*** essentially mirrors the pseudo-code of removing an element at the head of the list, except that we maintain a local reference to the element that is stored at the node that is being removed, and we return that element to the caller of ***pop***.
- A complete implementation of our ***LinkedStack*** class is given below.

Implementing a Stack with a Singly Linked List

```
class LinkedStack:
  """LIFO Stack implementation using a singly linked list for storage."""
  #------------------------- nested 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
    #------------------------------ stack methods ------------------------------
  def __init__(self):
   """Create an empty stack."""
   self._head = None # reference to the head node
   self._size = 0 # number of stack elements

  def len(self):
   """Return the number of elements in the stack."""
   return self._size

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

Implementing a Stack with a Singly Linked List

```
def push(self, e):
 """Add element e to the top of the stack."""
 self._head = self._Node(e, self._head) # create and link a new node
 self._size += 1

def top(self):
 """Return (but do not remove) the element at the top of the stack.
 Raise Empty exception if the stack is empty."""
 if self.is_empty( ):
  raise Empty('Stack is empty')
 return self._head._element # top of stack is at head of list

def pop(self):
 """Remove and return the element from the top of the stack (i.e., LIFO).
 Raise Empty exception if the stack is empty."""
 if self.is_empty( ):
  raise Empty('Stack is empty')
 answer = self._head._element
 self._head = self._head._next # bypass the former top node
 self._size -= 1
 return answer
```


Implementing a Stack with a Singly Linked List

- The following code exercises the operations of the ***LinkedStack*** class.

```
def main():
 S = LinkedStack()
 S.push(5)              # contents: [5]
 S.push(3)              # contents: [5, 3]
 print(S.len())         # contents: [5, 3];      outputs 2
 print(S.pop( ))        # contents: [5];         outputs 3
 print(S.is_empty())    # contents: [5];         outputs False
 print(S.pop( ))        # contents: [ ];         outputs 5
 print(S.is_empty())    # contents: [ ];         outputs True
 S.push(7)              # contents: [7]
 S.push(9)              # contents: [7, 9]
 print(S.top( ))        # contents: [7, 9];      outputs 9
 S.push(4)              # contents: [7, 9, 4]
 print(S.len())         # contents: [7, 9, 4];   outputs 3
 print(S.pop( ))        # contents: [7, 9];      outputs 4
 S.push(6)              # contents: [7, 9, 6]
if __name__ == '__main__':
 main()
```

Implementing a Queue with a Singly Linked List

- Now, we will implement the ***queue ADT*** with a ***singly linked list.***
- Because we need to perform operations on both ends of the queue, we will explicitly maintain both a ***_head*** reference and a ***_tail*** reference as instance variables for each queue.
- The natural orientation for a queue is to align the ***front*** of the queue with the ***_head*** of the list, and the ***back*** of the queue with the ***_tail*** of the list, because we must be able to ***enqueue*** elements at the ***back***, and ***dequeue*** them from the ***front***
- Many aspects of the LinkedQueue class implementation are similar to that of the LinkedStack class, such as the definition of the nested _Node class.

Implementing a Queue with a Singly Linked List

- The implementation of ***dequeue*** for ***LinkedQueue*** is similar to that of pop for ***LinkedStack***, as both remove the head of the linked list.
  - The only difference is that the queue must accurately maintain the ***tail*** reference.
  - In general, an operation at the ***head*** has no effect on the tail, but when ***dequeue*** is invoked on a queue with one element, we are simultaneously removing the ***tail*** of the list. Therefore, ***self._tail*** is set to ***None***.
- In the implementation of ***enqueue***, the newest node always becomes the new tail.
  - If the new node is the only node in the list, it also becomes the new ***head***
  - Otherwise the new node must be linked immediately after the existing ***tail*** node.
- The implementation of a LinkedQueue class is given below.

Implementing a Queue with a Singly Linked List

```
class LinkedQueue:
 """FIFO queue implementation using a singly linked list for storage."""
 class _Node:
  """Lightweight, nonpublic class for storing a singly linked node."""
        (omitted here; identical to that of LinkedStack Node)
 #------------------------------ queue methods ------------------------------
 def __init__(self):
  """Create an empty queue."""
  self._head = None
  self._tail = None
  self._size = 0 # number of queue elements

 def len(self):
  """Return the number of elements in the queue."""
  return self._size

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


Implementing a Queue with a Singly Linked List

```
def first(self):
  """Return (but do not remove) the element at the front of the queue."""
  if self.is_empty( ):
    raise Empty('Queue is empty')
  return self._head._element # front aligned with head of list

 def dequeue(self):
  """Remove and return the first element of the queue (i.e., FIFO).
  Raise Empty exception if the queue is empty."""
  if self.is_empty( ):
    raise Empty('Queue is empty')
  answer = self._head._element
  self._head = self._head._next
  self._size -= 1
  if self.is_empty( ): # special case as queue is empty
    self._tail = None # removed head had been the tail
  return answer
```

Implementing a Queue with a Singly Linked List

```
def enqueue(self, e):
  """Add an element to the back of queue."""
  newest = self._Node(e, None) # node will be new tail node
  if self.is_empty( ):
    self._head = newest # special case: previously empty
  else:
    self._tail._next = newest
  self._tail = newest # update reference to tail node
  self._size += 1
```

- The following code exercises the operations of the LinkedQueue class.

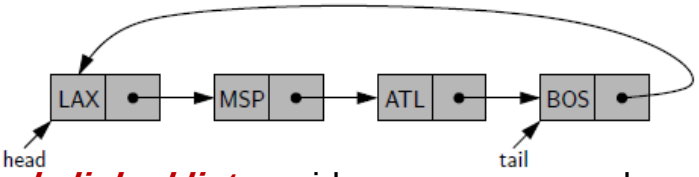
```
Q = LinkedQueue()
Q.enqueue(5)          # contents: [5]
Q.enqueue(3)          # contents: [5, 3]
print(Q.len())        # contents: [5, 3];      outputs 2
print(Q.dequeue())    # contents: [3];         outputs 5
print(Q.is_empty())   # contents: [3];         outputs False
print(Q.dequeue())    # contents: [ ];         outputs 3
print(Q.is_empty())   # contents: [ ];         outputs True
Q.enqueue(7)          # contents: [7]
Q.enqueue(9)          # contents: [7, 9]
print(Q.first())      # contents: [7, 9];      outputs 7
Q.enqueue(4)          # contents: [7, 9, 4]
print(Q.len())        # contents: [7, 9, 4];   outputs 3
print(Q.dequeue())    # contents: [9, 4];      outputs 7
```

25

Circularly Linked Lists

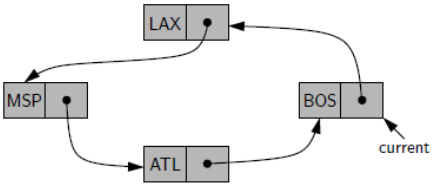
- In a ***circularly linked list***, the ***next*** reference of the ***tail*** points back to the ***head*** of the list, as shown in Figure (a).

Figure (a)



- A ***circularly linked list*** provides a more general model than a standard linked list for data sets that are cyclic, that is, which do not have any particular notion of a beginning and end.
- Figure (b) provides a more symmetric illustration of the same circular list structure of Figure (a).

Figure (b)



26

Circularly Linked Lists

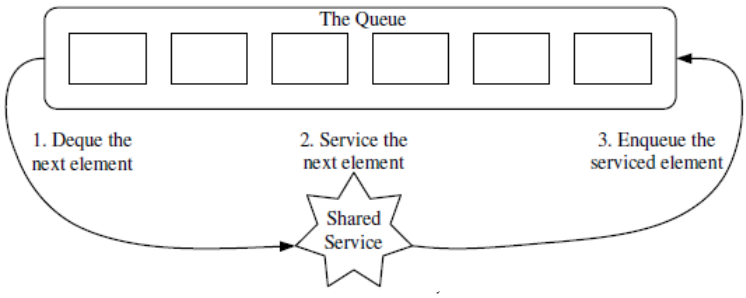
- A circular view similar to Figure (b) could be used, for example, to describe the order in which players take turns during a game.
- Even though a circularly linked list has no beginning or end, we must maintain a reference to a particular node in order to make use of the list.
- We use the identifier ***current*** to describe such a designated node.
- By setting **current = current.next**, we can effectively advance through the nodes of the list.

➢ ***Round-Robin Schedulers***

- As an application of ***circularly linked lists***, we consider a ***round-robin scheduler***, which iterates through a collection of elements in a circular fashion and "services" each element by performing a given action on it.

Circularly Linked Lists

- For example, ***round-robin scheduling*** is often used to allocate slices of CPU time to various applications running concurrently on a computer.
- A ***round-robin scheduler*** could be implemented with the general ***queue ADT***, by repeatedly performing the following steps on queue Q (see the figure below):
  1. e = Q.dequeue()
  2. Service element e
  3. Q.enqueue(e)



Circularly Linked Lists

- If we use the ***LinkedQueue*** class for such an application, there is unnecessary effort in the combination of a ***dequeue*** operation followed soon after by an ***enqueue*** of the same element.
- If we use a ***circularly linked list***, the effective transfer of an item from the ***head*** of the list to the ***tail*** of the list can be accomplished by advancing a reference that marks the boundary of the queue.
- We will next provide an implementation of a ***CircularQueue*** class that supports the entire ***queue ADT***, together with an additional method, ***rotate( )***, that *moves the first element of the queue to the back*.
- With this operation, a ***round-robin schedule*** can be implemented more efficiently by repeatedly performing the following steps:
  1. Service element *Q*.front()
  2. *Q*.rotate()

29

Implementing a Queue with a Circularly Linked List

- To implement the ***queue ADT*** using a ***circularly linked list***, we rely on the intuition of Figure (a), in which the ***queue*** has a ***head*** and a ***tail***, but with the next reference of the tail linked to the head.
- Given such a model, there is no need to explicitly store references to both the head and the tail; as long as we keep a reference to the tail, we can always find the head by following the tail's next reference.
- In implementing a ***CircularQueue*** class based on this model, we use only two instance variables:
  - ***_tail***, which is a reference to the tail node (or None when empty),
  - ***_size***, which is the current number of elements in the queue.
- When an operation involves the ***front*** of the queue, we recognize ***self._tail._next*** as the ***head*** of the queue.

Data Structures in Python - Prof. Moheb Ramzy Girgis Dept. of Computer Science - Faculty of Science Minia University

30

Implementing a Queue with a Circularly Linked List

- When ***enqueue*** is called, a new node is placed just after the ***tail*** but before the current ***head***, and then the new node becomes the ***tail***.
- In addition to the traditional queue operations, the ***CircularQueue*** class supports a ***rotate*** method that more efficiently enacts the combination of removing the front element and reinserting it at the back of the queue.
  - With the circular representation, the ***rotate*** method sets self._tail = self._tail._next to make the old ***head*** become the new ***tail*** (with the node after the old ***head*** becoming the new ***head***).
- An implementation of a ***CircularQueue*** class is given below.

Implementing a Queue with a Circularly Linked List

```
class CircularQueue:
 """Queue implementation using circularly linked list for storage."""
 #------------------------- nested 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
 #-----------------------circular queue methods -----------------------
 def __init__ (self):
  """Create an empty queue."""
  self._tail = None # will represent tail of queue
  self._size = 0 # number of queue elements

 def len(self):
  """Return the number of elements in the queue."""
  return self._size

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

Implementing a Queue with a Circularly Linked List

```
def first(self):
 """Return (but do not remove) the element at the front of the queue.
 Raise Empty exception if the queue is empty. """
 if self.is_empty( ):
  raise Empty('Queue is empty')
 head = self._tail._next
 return head._element

def dequeue(self):
 """Remove and return the first element of the queue (i.e., FIFO).
 Raise Empty exception if the queue is empty. """
 if self.is_empty( ):
  raise Empty('Queue is empty')
 oldhead = self._tail._next
 if self._size == 1:       # removing only element
  self._tail = None        # queue becomes empty
 else:
  self._tail._next = oldhead._next          # bypass the old head
 self._size -= 1
 return oldhead._element
```

Girgis Dept. of Computer Science - Faculty of Science Minia University

33

Implementing a Queue with a Circularly Linked List

```
def enqueue(self, e):
 """Add an element to the back of queue."""
 newest = self._Node(e, None) # node will be new tail node
 if self.is_empty( ):
  newest._next = newest # initialize circularly
 else:
  newest._next = self._tail._next # new node points to head
  self._tail._next = newest # old tail points to new node
 self._tail = newest # new node becomes the tail
 self._size += 1

def rotate(self):
 """Rotate front element to the back of the queue."""
 if self._size > 0:
  self._tail = self._tail._next # old head becomes new tail
```

Data Structures in Python - Prof. Moheb Ramzy
Girgis Dept. of Computer Science - Faculty of Science Minia University

34

Implementing a Queue with a Circularly Linked List

- The following code exercises the operations of the ***CircularQueue*** class

```
Q = CircularQueue()
Q.enqueue(5)          # contents: [5]
Q.enqueue(3)          # contents: [5, 3]
print(Q.len())        # contents: [5, 3];      outputs 2
print(Q.dequeue())    # contents: [3];         outputs 5
print(Q.is_empty())   # contents: [3];         outputs False
print(Q.dequeue())    # contents: [ ];         outputs 3
print(Q.is_empty())   # contents: [ ];         outputs True
Q.enqueue(7)          # contents: [7]
Q.enqueue(9)          # contents: [7, 9]
print(Q.first())      # contents: [7, 9];      outputs 7
Q.enqueue(4)          # contents: [7, 9, 4]
print(Q.len())        # contents: [7, 9, 4];   outputs 3
Q.rotate()            # contents: [9, 4, 7]
print(Q.first())      # contents: [9, 4, 7];   outputs 9
Q.rotate()            # contents: [4, 7, 9]
print(Q.first())      # contents: [4, 7, 9];   outputs 4
print(Q.dequeue())    # contents: [7, 9];      outputs 4
```


Doubly Linked Lists

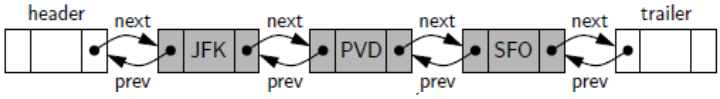
- In a ***singly linked list***, each node maintains a reference to the node that is immediately after it.
- As mentioned before, we can efficiently insert a node at either end of a singly linked list, and can delete a node at the head of a list, but we are unable to efficiently delete a node at the tail of the list, because we cannot determine the node that immediately *precedes* the node to be deleted.
- To provide greater symmetry, we 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*.
- Such a structure is known as a ***doubly linked list***.
- These lists allow a greater variety of update operations, including insertions and deletions at arbitrary positions within the list.

Doubly Linked Lists

- We continue to use the term "***next***" for the reference to the node that follows another, and we introduce the term "***prev***" for the reference to the node that precedes it.

➢ ***Header and Trailer 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,
  - 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.
- ***Sentinels*** allow us to treat all insertions and deletions in a unified manner.
- A ***doubly linked list*** with such ***sentinels*** is shown below.



37

Doubly Linked Lists

- When using ***sentinel*** nodes, an empty list is initialized so that the ***next*** field of the ***header*** points to the ***trailer***, and the ***prev*** field of the ***trailer*** points to the ***header***.
- The remaining fields of the ***sentinels*** are irrelevant (presumably ***None***, in Python).
- For a nonempty list, the header's ***next*** will refer to a node containing the first real element of a sequence, and the trailer's ***prev*** references the node containing the last element of a sequence.

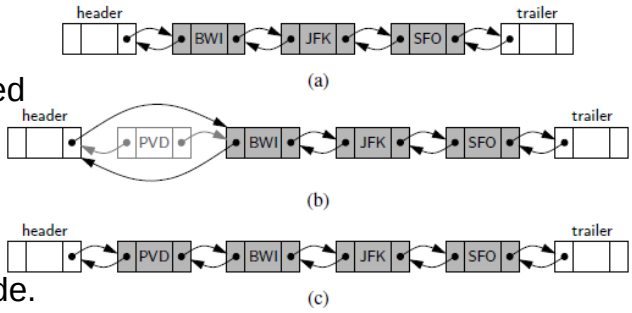
➢ ***Inserting and Deleting with a Doubly Linked List***

- Every ***insertion*** into our ***doubly linked list*** representation will take place between a pair of existing nodes.
- For example, when a new element is inserted at the front of the sequence, we will simply add the new node between the header and the node that is currently after the header. (See the following figure)

38

Doubly Linked Lists

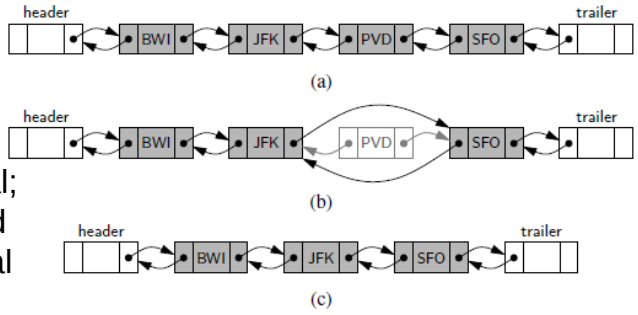
- The figure illustrates the addition of an element to the front of a doubly linked list with sentinels: (a) before the operation; (b) after creating the new node; (c) after linking the neighbors to the new node.



  - The ***deletion*** of a node proceeds in the opposite fashion of an insertion, as shown in the following figure.
  - The two neighbors of the node to be deleted are linked directly to each other, thereby bypassing the original node.
  - As a result, that node will no longer be considered part of the list and it can be reclaimed by the system.

Doubly Linked Lists

- The figure illustrates the removal of the element PVD from a doubly linked list: (a) before the removal; (b) after linking out the old node; (c) after the removal (and garbage collection).



  - Because of the use of ***sentinels***, the same implementation can be used when deleting the first or the last element of a sequence, because even such an element will be stored at a node that lies between two others.

Basic Implementation of a Doubly Linked List

- We begin by providing a preliminary implementation of a ***doubly linked list***, in the form of a class named ***_DoublyLinkedBase***.
- This class will not provide a coherent public interface for general use.
- Later, we will develop a public class that inherits from our ***_DoublyLinkedBase*** class to provide more coherent abstractions:
  - A ***LinkedDeque*** class that implements the ***double-ended queue ADT***, which only supports operations at the ends of the queue, so there is no need for a user to identify an interior position within the list.

Basic Implementation of a Doubly Linked List

- The ***_DoublyLinkedBase*** class relies on the use of a nonpublic ***_Node*** class that is similar to that for a singly linked list, except that the doubly linked version includes a ***_prev*** attribute, in addition to the ***_next*** and ***_element*** attributes, as shown below.

```
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
```

- The remainder of the ***_DoublyLinkedBase*** class is given below:

Basic Implementation of a Doubly Linked List

```
class _DoublyLinkedBase:
 """A base class providing a doubly linked list representation."""“
#-------------------------------------------------------------------------------------------
 class _Node:
  """Lightweight, nonpublic class for storing a doubly linked node."""
        (omitted here; see previous code fragment)
 #-------------------------------------------------------------------------------------------
 def __init__ (self):
  """Create an empty list."""
  self._header = self._Node(None, None, None)
  self._trailer = self._Node(None, None, None)
  self._header._next = self._trailer # trailer is after header
  self._trailer._prev = self._header # header is before trailer
  self._size = 0 # number of elements

 def len(self):
  """Return the number of elements in the list."""
  return self._size

 def is_empty(self):
  """Return True if list is empty."""
  return self._size == 0
```

43

Basic Implementation of a Doubly Linked List

```
def _insert_between(self, e, predecessor, successor):
  """Add element e between two existing nodes and return new node."""
  newest = self._Node(e, predecessor, successor) # linked to neighbors
  predecessor._next = newest
  successor._prev = newest
  self._size += 1
  return newest

 def _delete_node(self, node):
  """Delete nonsentinel node from the list and return its element."""
  predecessor = node._prev
  successor = node._next
  predecessor._next = successor
  successor._prev = predecessor
  self._size -= 1
  element = node._element # record deleted element
  node._prev = node._next = node._element = None # deprecate node
  return element # return deleted element
```

Basic Implementation of a Doubly Linked List

- The ***constructor*** instantiates the two ***sentinel*** nodes and links them directly to each other.
- The class maintains a ***_size*** member and provide public support for ***len*** and ***is_empty*** so that these behaviors can be directly inherited by the subclasses.
- The other two methods of the class are the nonpublic utilities, ***_insert_between*** and ***_delete_node***.
- These provide generic support for insertions and deletions, respectively, but require one or more node references as parameters.
- The ***_insert_between*** method:
  - Creates a new node, and initializes its fields to link to the specified neighboring nodes.
  - Updates the fields of the neighboring nodes to include the new node in the list.
  - Returns a reference to the newly created node.

45

Basic Implementation of a Doubly Linked List

- The ***_delete_node*** method:
  - Links the neighbors of the node to be deleted directly to each other, thereby bypassing the deleted node from the list.
  - Records the element of the deleted node to be returned.
  - Resets the ***_prev***, ***_next***, and ***_element*** fields of the deleted node to ***None***. This may help Python's garbage collection.
  - Returns the deleted element.

Implementing a Deque with a Doubly Linked List

- The ***double-ended queue (deque) ADT*** was introduced in Lecture 7.
- We will provide an implementation of a ***LinkedDeque*** class that inherits from the ***_DoublyLinkedBase*** class.
- We do not provide an explicit ***__init__*** method for the ***LinkedDeque*** class, as the inherited version of that method suffices to initialize a new instance.
- We also use the inherited methods ***len*** and ***is_empty***.
- With the use of ***sentinels***, the key to this implementation is to remember that:
  - the first element of the ***deque*** is stored in the node just *after* the ***header*** (assuming the ***deque*** is nonempty).
  - the last element of the ***deque*** is stored in the node just *before* the ***trailer***.

Implementing a Deque with a Doubly Linked List

- We use the inherited ***_insert_between*** method to insert at either end of the ***deque***.
  - To ***insert*** an element at the ***front*** of the ***deque***, we place it between the header and the node just after the header.
  - To ***insert*** an element at the ***end*** of ***deque***, we place it immediately before the trailer node.
  - When the ***deque*** is empty, the new node is placed between the two ***sentinels***.
- When deleting an element from a nonempty ***deque***, we use the inherited ***_delete_node*** method, knowing that the designated node is assured to have neighbors on each side.

Implementing a Deque with a Doubly Linked List

```
class LinkedDeque(_DoublyLinkedBase): # note the use of inheritance
 """Double-ended queue implementation based on a doubly linked list."""

 def first(self):
  """Return (but do not remove) the element at the front of the deque."""
  if self.is_empty( ):
   raise Empty("Deque is empty")
  return self._header._next._element # real item just after header

 def last(self):
  """Return (but do not remove) the element at the back of the deque."""
  if self.is_empty( ):
   raise Empty("Deque is empty")
  return self._trailer._prev._element # real item just before trailer

 def insert_first(self, e):
  """Add an element to the front of the deque."""
  self._insert_between(e, self._header, self._header._next) # after header

 def insert_last(self, e):
  """Add an element to the back of the deque."""
  self._insert_between(e, self._trailer._prev, self._trailer) # before trailer
```

49

Implementing a Deque with a Doubly Linked List

```
 def delete_first(self):
  """Remove and return the element from the front of the deque.
  Raise Empty exception if the deque is empty. """
  if self.is_empty( ):
   raise Empty("Deque is empty")
  return self._delete_node(self._header._next) # use inherited method

 def delete_last(self):
  """Remove and return the element from the back of the deque.
  Raise Empty exception if the deque is empty. """
  if self.is_empty( ):
   raise Empty("Deque is empty")
  return self._delete_node(self._trailer._prev) # use inherited method
```

Implementing a Deque with a Doubly Linked List

- The following code exercises the operations of the ***LinkedDeque*** class.

```
D = LinkedDeque()
D.insert_last(5)          # contents: [5]
D.insert_first(3)         # contents: [3, 5]
D.insert_first(7)         # contents: [7, 3, 5]
print(D.len())            # contents: [7, 3, 5];      outputs 3
print(D.first())          # contents: [7, 3, 5];      outputs 7
print(D.delete_last())  # contents: [7, 3];         outputs 5
print(D.last())           # contents: [7, 3];         outputs 3
print(D.len())            # contents: [7, 3];         outputs 2
print(D.delete_last())    # contents: [7];            outputs 3
print(D.delete_last())    # contents: [];             outputs 7
print(D.is_empty())       # contents: [];             outputs True
D.insert_first(6)         # contents: [6]
print(D.last())           # contents: [6];            outputs 6
D.insert_first(8)         # contents: [8, 6]
print(D.is_empty())       # contents: [8, 6];         outputs False
print(D.last())           # contents: [8, 6];         outputs 6
print(D.delete_first())   # contents: [6];            outputs 8
print(D.first())          # contents: [6];            outputs 6
```

51