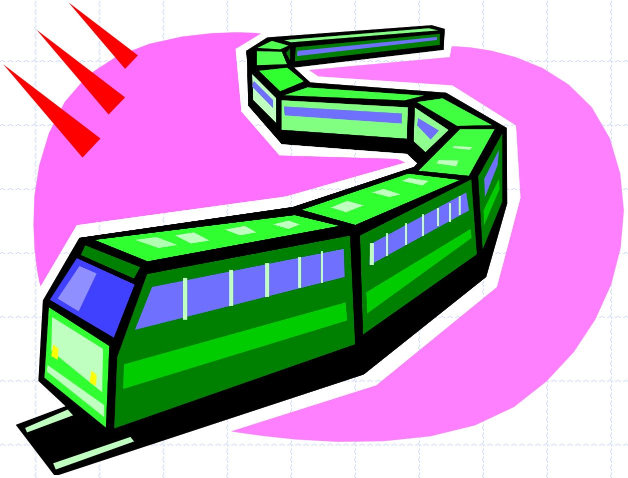
Doubly-Linked Lists



Assignment #6: :Linked Lists

- R-7.2 Describe a good algorithm for concatenating two singly linked lists L and M, given only references to the first node of each list, into a single list L` that contains all the nodes of L followed by all the nodes of M.
- R-7.5 Implement a function that counts the number of nodes in a circularly linked list.(an implementation of the __len__ method for our CircularQueue class that does not presume use of a precalculated _size member.)
- R-7.7 Our CircularQueue class of Section 7.2.2 provides a rotate( ) method that has semantics equivalent to Q.enqueue(Q.dequeue( )), for a nonempty queue. Implement such a method for the LinkedQueue class of Section 7.1.2 without the creation of any new nodes.

R-7.2 concatenating two singly linked lists

R-7.2.py - C:\course\cs202\ass6\R-7.2.py (3.4.3)

File Edit Format Run Options Window Help

```python
def concatenate(L,M):
  walk = L._head
  while walk._next is not None:
    walk = walk._next
  walk._next = M._head
  M._head = None

if __name__ == '__main__':
  L = LinkedQueue(); M = LinkedQueue()
  L.enqueue('l1');L.enqueue('l2');L.enqueue('l3')
  M.enqueue('m1');M.enqueue('m2');M.enqueue('m3')
  concatenate(L,M)
  print(' '.join(L))
  #for e in L:
  #  print(e)
```

Ln: 25 Col: 0

Window Help

```
3e601, Feb 24 2015, 22:44:40) [MSC v.1600 64 bit (AM
or "license()" for more information.
>>> ================================ RESTART ================================
>>>
Traceback (most recent call last):
** IDLE Internal Exception:
  File "C:\Python34\lib\idlelib\run.py", line 353, in runcode
    exec(code, self.locals)
  File "C:\course\cs202\2015\ref\Codes\ch07\linked_queue.py", line 96, in <modul
e>
    for e in L:
TypeError: 'LinkedQueue' object is not iterable
>>>
```

Ln: 12 Col: 4

linked_queue.py - C:\course\cs202\2015\ref\Codes\ch07\linked_queue.py (3.4.3)

File Edit Format Run Options Window Help

```
from exceptions import Empty
class LinkedQueue:
  """FIFO queue 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):
      self._element = element
      self._next = next
  #------------------------------- qu
  def __init__(self):
    """Create an empty queue."""
    self._head = None
    self._tail = None
    self._size = 0
  def __len__(self):
    """Return the number of elements
    return self._size
  # added by YHF
  def __iter__(self):
    """Generate a forward iteration 
    cursor = self._head
    while cursor is not None:
      yield cursor._element
      cursor = cursor._next
  # added end
  def is_empty(self):
    """Return True if the queue is e
    return self._size == 0
```

Python 3.4.3 Shell

File Edit Shell Debug Options Window Help

```
>>>
11 12 13 m1 m2 m3
>>> 
```

Ln: 6 Col: 4

linked_queue.py - C:\course\cs202\2015\ref\Codes\ch07\linked_queue.py (3.4.3)

File Edit Format Run Options Window Help

```
  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')
    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
  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
```

Ln: 50 Col: 21

R-7.5 count the number of nodes in a circularly link list

```python
from exceptions import Empty

class CircularQueue:
  """Queue implementation using circularly l

  #-----------------------------------------
  # nested _Node class
  class _Node:
    """Lightweight, nonpublic class for stor
    __slots__ = '_element', '_next'

    def __init__(self, element, next):
      self._element = element
      self._next = next

  # end of _Node class
  #-----------------------------------------

  def __init__(self):
    """Create an empty queue."""
    self._tail = None                       # 
    self._size = 0                          # 

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

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

  def first(self):
    """Return (but do not remove) the elemen

    Raise Empty exception if the queue is em
    """
    if self.is_empty():
      raise Empty('Queue is empty')
    head = self._tail._next
    return head._element
```

R7.5.py - C:\course\cs202\2015\ref\Codes\ch07\R7.5.py (3.4.3)

File Edit Format Run Options Window Help

```python
  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

  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
```

Ln: 72 Col: 0

*R7.5.py - C:\course\cs202\2015\ref\Codes\ch07\R7.5.py (3.4....

File Edit Format Run Options Window Help

```python
  def __mylen__(self):
    if self._tail == None:
      return 0

    walk = self._tail._next
    count = 1
    while walk != self._tail:
      count += 1
      walk = walk._next
    return count

if __name__ == '__main__':
  CL = CircularQueue()
  CL.enqueue('cl1'); CL.enqueue('cl2'); CL.enqueue('cl3')
  print(CL.__mylen__())
  ZeroCL = CircularQueue()
  print(ZeroCL.__mylen__())
```

Ln: 89 Col: 27

Python 3.4.3 Shell

File Edit Shell Debug Options Window Help

```
>>>
3
0
>>>
```

Ln: 7 Col: 4

R-7.7 provides a rotate() method for the LinkedQueue

```python
from exceptions import Empty

class LinkedQueue:
  """FIFO queue implementation using a singly linked list

  #-------------------------- nested _Node class ---------
  class _Node:
    """Lightweight, nonpublic class for storing a singly l
    __slots__ = '_element', '_next'          # streamline m

    def __init__(self, element, next):
      self._element = element
      self._next = next

  #------------------------------- queue methods ---------
  def __init__(self):
    """Create an empty queue."""
    self._head = None
    self._tail = None
    self._size = 0                           # number of qu

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

  # added by YHF
  def __iter__(self):
    """Generate a forward iteration of the elements of the
    cursor = self._head
    while cursor is not None:
      yield cursor._element
      cursor = cursor._next
  # added end

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

  def first(self):
    """Return (but do not remove) the element at the front

    Raise Empty exception if the queue is empty.
    """
    if self.is_empty():
      raise Empty('Queue is empty')
    return self._head._element               # front aligne
```

*R-7.7.py - C:\course\cs202\ass6\R

File Edit Format Run Options

```python
  def dequeue(self):
    """Remove and return th

    Raise Empty exception i
    """
    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
      self._tail = None                              # removed head
    return answer

  def enqueue(self, e):
    """Add an element to the back of queue."""
    newest = self._Node(e, None)                     # node will be
    if self.is_empty():
      self._head = newest                            # special case:
    else:
      self._tail._next = newest
    self._tail = newest                              # update refere
    self._size += 1
  #==added==
  def rotate(self):
    """semantics equivalent to Q.enqueue(Q.dequeue( ))"""
    if not self.is_empty():
      self._tail._next = self._head
      self._head = self._head._next
      self._tail = self._tail._next
      self._tail._next = None

if __name__ == '__main__':
  L = LinkedQueue()
  L.enqueue('11');L.enqueue('12');L.enqueue('13')
  print(' '.join(L))
  L.rotate()
  print(' '.join(L))
```

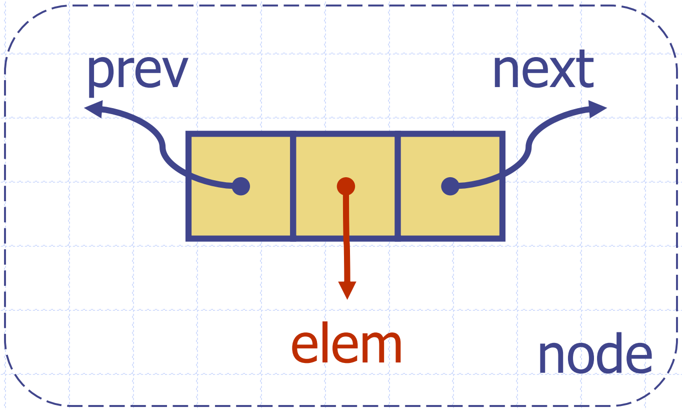
Python 3.4.3 Shell

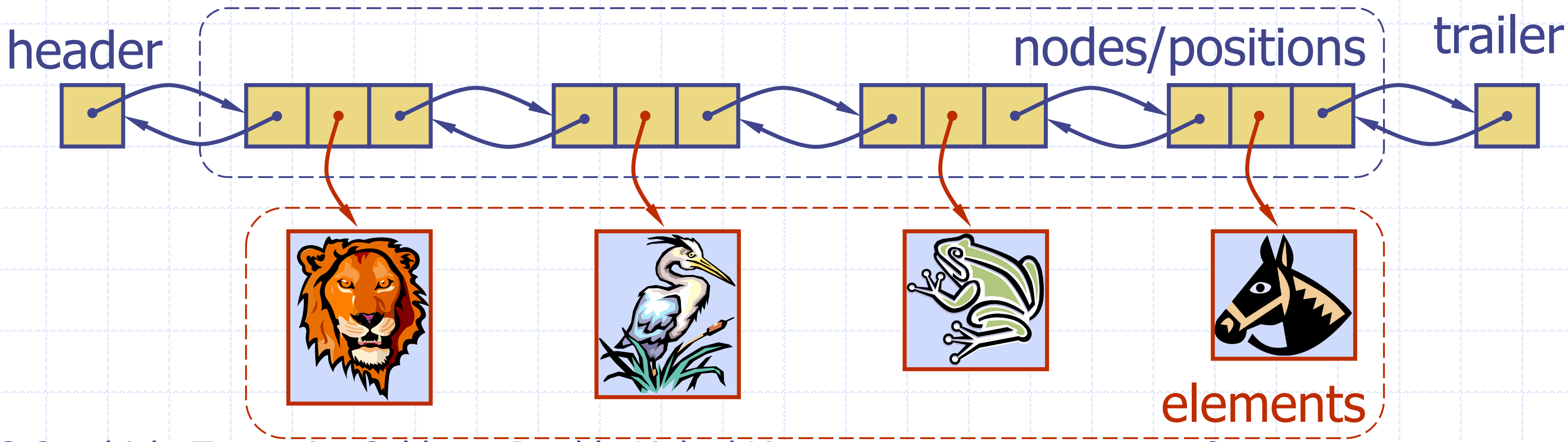
File Edit Shell Debug Options Window H

```
>>>
11 12 13
12 13 11
>>>
```

Doubly Linked List

- A doubly linked list provides a natural implementation of the Node List ADT
- Nodes implement Position and store:
  - element
  - link to the previous node
  - link to the next node
- Special trailer and header nodes



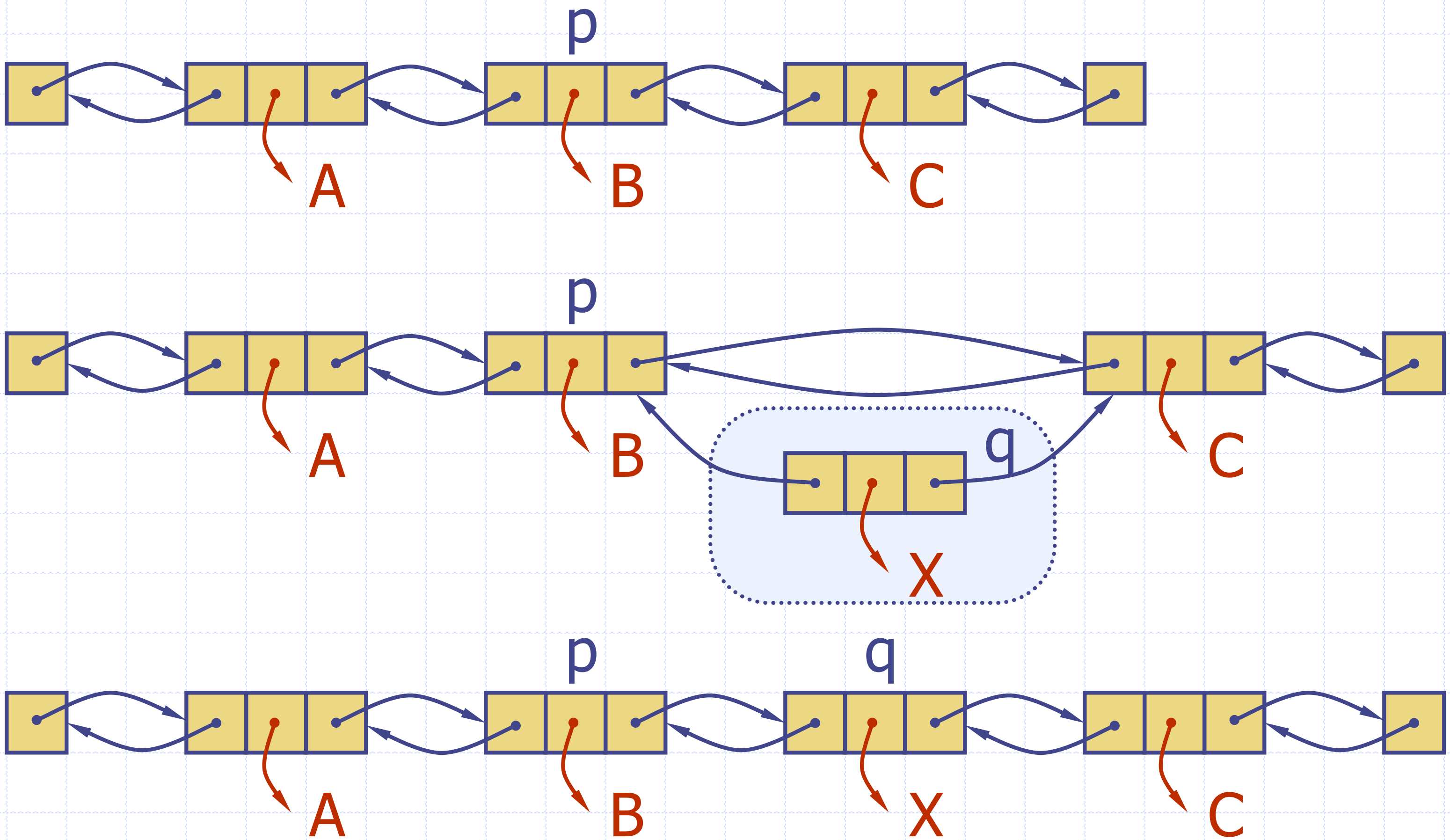


limitations that stem from the asymmetry of a singly linked list

- we can efficiently insert a node at either end of a singly linked list, and can delete a node at the head of a list,
  - but we are unable to efficiently delete a node at the tail of the list.
- More generally, we cannot efficiently delete an arbitrary node from an interior position of the list if only given a reference to that node,
  - because we cannot determine the node that immediately *precedes* the node to be deleted (yet, that node needs to have its next reference updated).

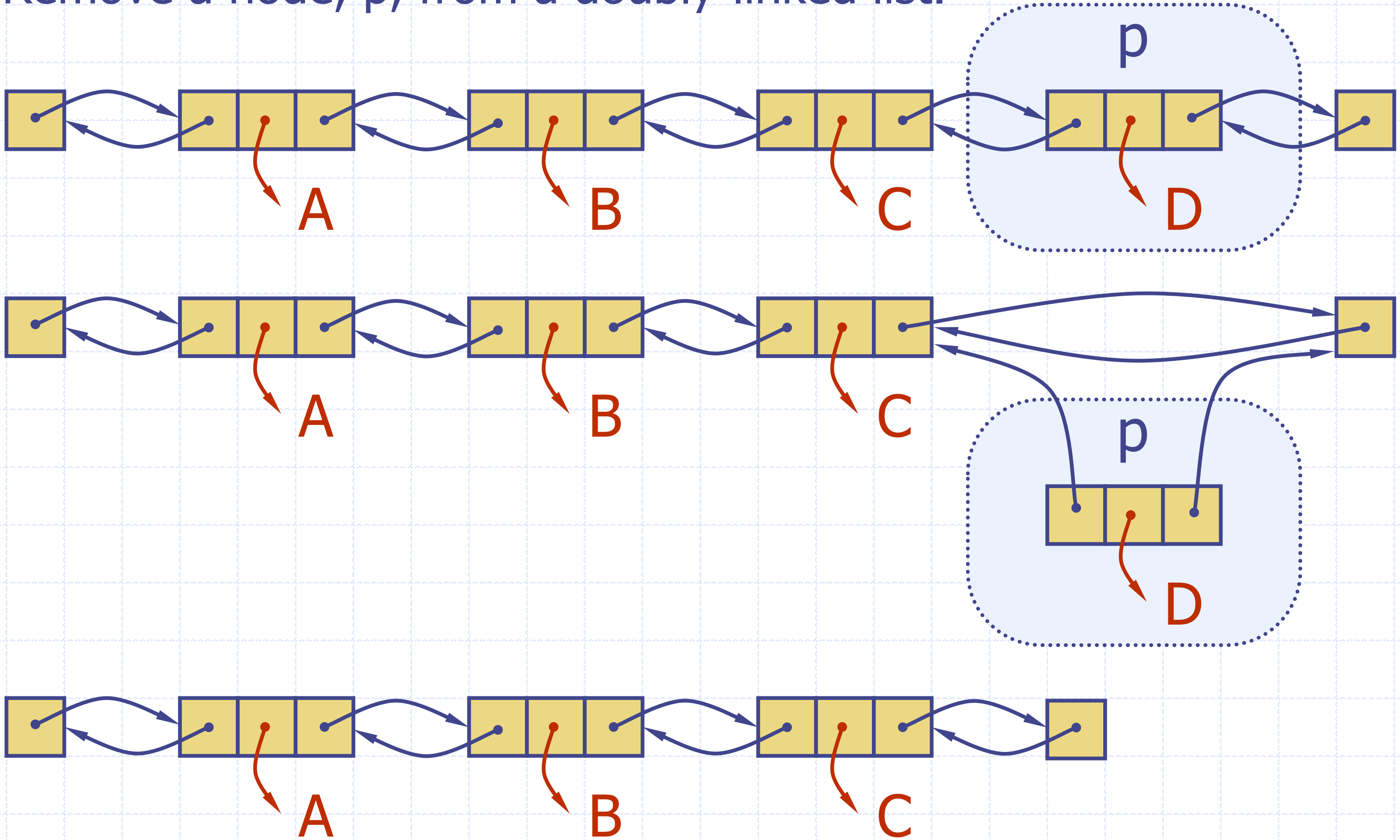
Insertion

- Insert a new node, q, between p and its successor.



Deletion

- Remove a node, p, from a doubly-linked list.



Doubly-Linked List in Python

```
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

  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
```

Performance

- In a doubly linked list
  - The space used by a list with n elements is $O(n)$
  - The space used by each position of the list is $O(1)$
  - All the standard operations of a list run in $O(1)$ time

Implementing a Deque with a Doubly Linked List

linked_deque.py - C:\course\cs202\2015\ref\Codes\ch07\linked_deque.py (3.4.3)

File Edit Format Run Options Window Help

```python
from doubly_linked_base import _DoublyLinkedBase
from exceptions import Empty

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 el

    Raise Empty exception if the deque i
    """
    if self.is_empty():
      raise Empty("Deque is empty")
    return self._header._next._element

  def last(self):
    """Return (but do not remove) the el

    Raise Empty exception if the deque i
    """
    if self.is_empty():
      raise Empty("Deque is empty")
    return self._trailer._prev._element
```

linked_deque.py - C:\course\cs202\2015\ref\Codes\ch07\linked_deque.py (3.4.3)

File Edit Format Run Options Window Help

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

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

  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
```

Ln: 49

The Positional List ADT

- What if a waiting customer decides to hang up before reaching the front of the customer service queue? Or what if someone who is waiting in line to buy tickets allows a friend to "cut" into line at that position?
- indices are not a good abstraction for describing a local position in some applications, because the index of an entry changes over time due to insertions or deletions that happen earlier in the sequence.

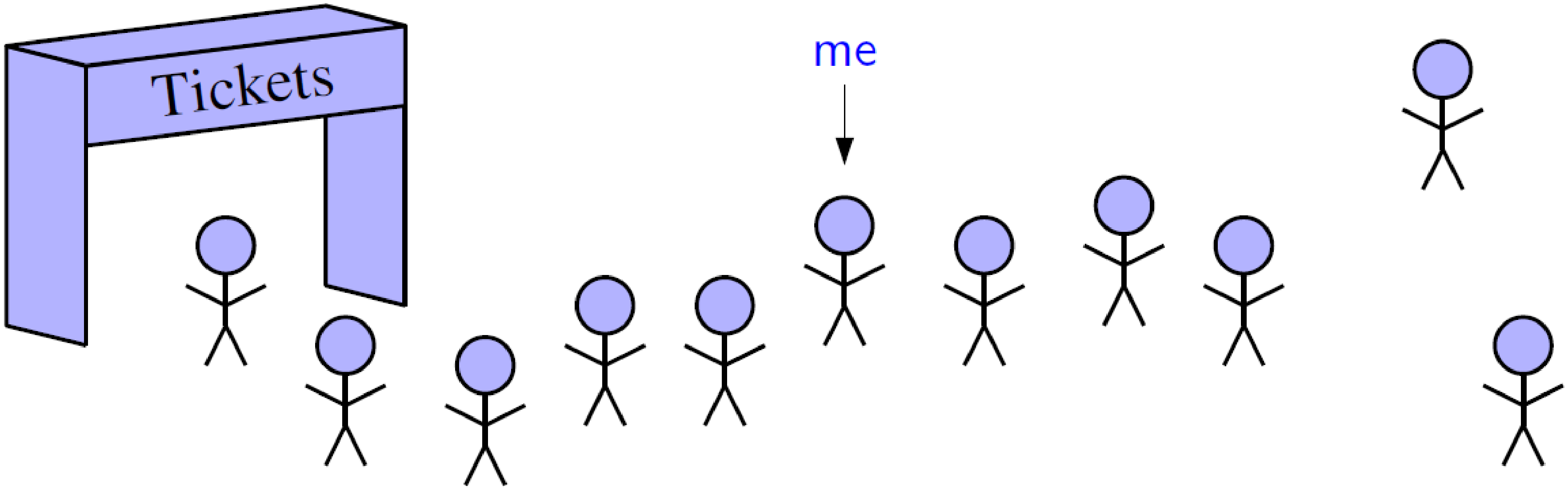


Figure 7.14: We wish to be able to identify the position of an element in a sequence without the use of an integer index.

As another example

- a text document can be viewed as a long sequence of characters.
- A word processor uses the abstraction of a **cursor** to describe a position within the document without explicit use of an integer index,
  - allowing operations such as "delete the character at the cursor" or "insert a new character just after the cursor."
- Furthermore, we may be able to refer to an inherent position within a document,
  - such as the beginning of a particular section, without relying on a character index (or even a section number) that may change as the document evolves.

Positional List

- To provide for a general abstraction of a sequence of elements with the ability to identify the location of an element, we define a **positional list** ADT.
- A position acts as a marker or token within the broader positional list.
- A position p is unaffected by changes elsewhere in a list; the only way in which a position becomes invalid is if an explicit command is issued to delete it.
- A position instance is a simple object, supporting only the following method:
  - p.element( ): Return the element stored at position p.

Positional Accessor Operations

L.first( ): Return the position of the first element of L, or None if L is empty.

L.last( ): Return the position of the last element of L, or None if L is empty.

L.before(p): Return the position of L immediately before position p, or None if p is the first position.

L.after(p): Return the position of L immediately after position p, or None if p is the last position.

L.is_empty( ): Return True if list L does not contain any elements.

len(L): Return the number of elements in the list.

iter(L): Return a forward iterator for the *elements* of the list. See Section 1.8 for discussion of iterators in Python.

Positional Update Operations

L.add_first(e): Insert a new element e at the front of L, returning the position of the new element.

L.add_last(e): Insert a new element e at the back of L, returning the position of the new element.

L.add_before(p, e): Insert a new element e just before position p in L, returning the position of the new element.

L.add_after(p, e): Insert a new element e just after position p in L, returning the position of the new element.

L.replace(p, e): Replace the element at position p with element e, returning the element formerly at position p.

L.delete(p): Remove and return the element at position p in L, invalidating the position.

Example 7.1: *The following table shows a series of operations on an initially empty positional list L. To identify position instances, we use variables such as p and q. For ease of exposition, when displaying the list contents, we use subscript notation to denote its positions.*

Operation	Return Value	L
L.add_last(8)	p	8_p
L.first()	p	8_p
L.add_after(p, 5)	q	$8_p, 5_q$
L.before(q)	p	$8_p, 5_q$
L.add_before(q, 3)	r	$8_p, 3_r, 5_q$
r.element()	3	$8_p, 3_r, 5_q$
L.after(p)	r	$8_p, 3_r, 5_q$
L.before(p)	None	$8_p, 3_r, 5_q$
L.add_first(9)	s	$9_s, 8_p, 3_r, 5_q$
L.delete(L.last())	5	$9_s, 8_p, 3_r$
L.replace(p, 7)	8	$9_s, 7_p, 3_r$

Positional List in Python (1/3)

positional_list.py - C:\course\cs202\2015\ref\Codes\ch07\positional_list.py (3.4.3)

File Edit Format Run Options Window Help

```python
from doubly_linked_base import _DoublyLinkedBase

class PositionalList(_DoublyLinkedBase):
  """A sequential container of elements allowing positional access."""

  #-------------------------- nested Position class --------------------------
  class Position:
    """An abstraction representing the location of a single element.

    Note that two position instaces may represent the same inherent
    location in the list.  Therefore, users should always rely on
    syntax 'p == q' rather than 'p is q' when testing equivalence of
    positions.
    """

    def __init__(self, container, node):
      """Constructor should not be invoked by user."""
      self._container = container
      self._node = node

    def element(self):
      """Return the element stored at this Position."""
      return self._node._element

    def __eq__(self, other):
      """Return True if other is a Position representing the same location."""
      return type(other) is type(self) and other._node is self._node

    def __ne__(self, other):
      """Return True if other does not represent the same location."""
      return not (self == other)               # opposite of __eq__

  #------------------------------- utility methods -------------------------------
  def _validate(self, p):
    """Return position's node, or raise appropriate error if invalid."""
    if not isinstance(p, self.Position):
      raise TypeError('p must be proper Position type')
    if p._container is not self:
      raise ValueError('p does not belong to this container')
    if p._node._next is None:                  # convention for deprecated nodes
      raise ValueError('p is no longer valid')
    return p._node
```

Ln: 42 Col: 18

Positional List in Python(2/2)

```
#------------------------------ utility method ------------------------------
def _make_position(self, node):
  """Return Position instance for given node (or None if sentinel)."""
  if node is self._header or node is self._trailer:
    return None                                    # boundary violation
  else:
    return self.Position(self, node)               # legitimate position

#------------------------------ accessors ------------------------------
def first(self):
  """Return the first Position in the list (or None if list is empty)."""
  return self._make_position(self._header._next)

def last(self):
  """Return the last Position in the list (or None if list is empty)."""
  return self._make_position(self._trailer._prev)

def before(self, p):
  """Return the Position just before Position p (or None if p is first)."""
  node = self._validate(p)
  return self._make_position(node._prev)

def after(self, p):
  """Return the Position just after Position p (or None if p is last)."""
  node = self._validate(p)
  return self._make_position(node._next)

def __iter__(self):
  """Generate a forward iteration of the elements of the list."""
  cursor = self.first()
  while cursor is not None:
    yield cursor.element()
    cursor = self.after(cursor)
```

Positional List in Python(3/3)

```python
#------------------------------ mutators ------------------------------
# override inherited version to return Position, rather than Node
def _insert_between(self, e, predecessor, successor):
  """Add element between existing nodes and return new Position."""
  node = super()._insert_between(e, predecessor, successor)
  return self._make_position(node)

def add_first(self, e):
  """Insert element e at the front of the list and return new Position."""
  return self._insert_between(e, self._header, self._header._next)

def add_last(self, e):
  """Insert element e at the back of the list and return new Position."""
  return self._insert_between(e, self._trailer._prev, self._trailer)

def add_before(self, p, e):
  """Insert element e into list before Position p and return new Position."""
  original = self._validate(p)
  return self._insert_between(e, original._prev, original)

def add_after(self, p, e):
  """Insert element e into list after Position p and return new Position."""
  original = self._validate(p)
  return self._insert_between(e, original, original._next)

def delete(self, p):
  """Remove and return the element at Position p."""
  original = self._validate(p)
  return self._delete_node(original)      # inherited method returns element

def replace(self, p, e):
  """Replace the element at Position p with e.

  Return the element formerly at Position p.
  """
  original = self._validate(p)
  old_value = original._element           # temporarily store old element
  original._element = e                   # replace with new element
  return old_value                        # return the old element value
```

positional_list.py - C:\course\cs202\2015\ref\Codes\ch07\positional_list.py

File Edit Format Run Options Window Help

```python
if __name__ == '__main__':
  L = PositionalList()
  L.add_last(8)
  cursor = L.first()
  cursor = L.add_after(cursor,5)
  L.add_before(cursor,3)
  L.add_first(9)
  print(' '.join(str(x) for x in L))
  #for e in L:
    #print(e)
```

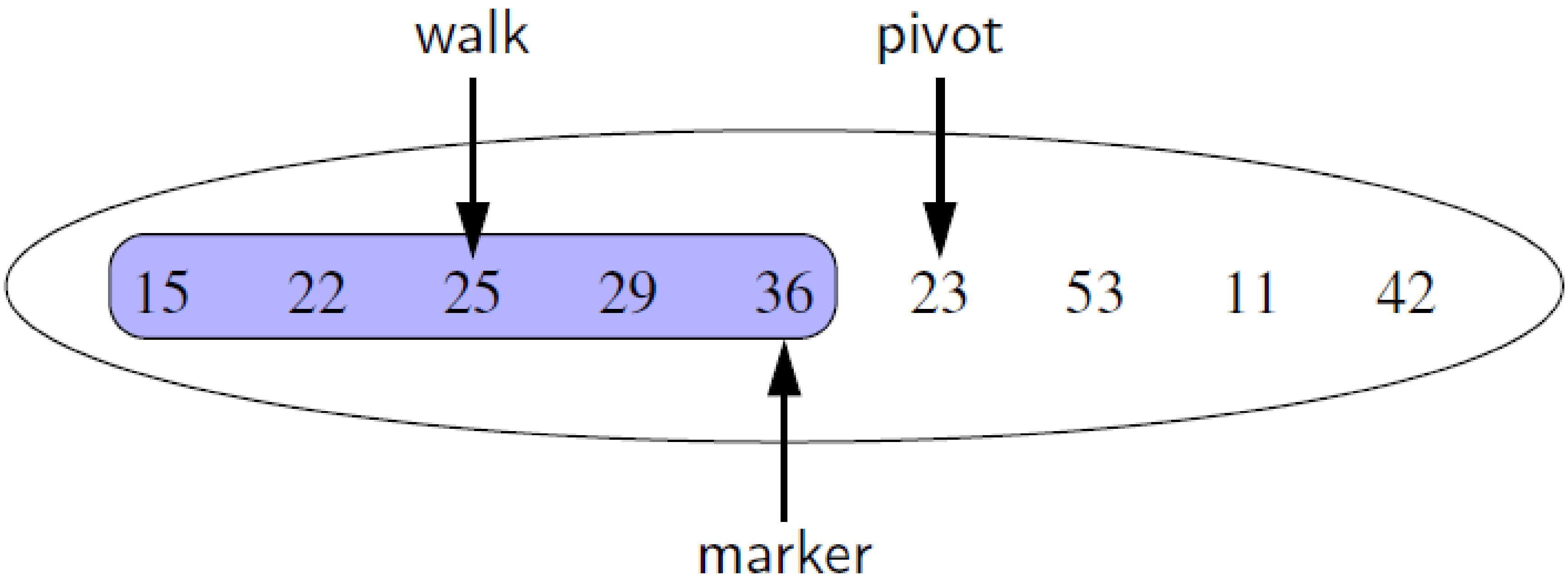
Python 3.4.3 Shell

File Edit Shell Debug Options Window Help

```
>>>
9 8 3 5
>>>
```

Ln: 6 Col: 4

7.5 Sorting a Positional List



- Each element is placed relative to a growing collection of previously sorted elements.

insertion_sort_positional.py - C:\course\cs202\2015\ref\Codes\ch07\insertion_sort_positi...

File Edit Format Run Options Window Help

```python
from positional_list import PositionalList

def insertion_sort(L):
  """Sort PositionalList of comparable elements into nondecreasing order."""
  if len(L) > 1:                    # otherwise, no need to sort it
    marker = L.first()
    while marker != L.last():
      pivot = L.after(marker)       # next item to place
      value = pivot.element()
      if value > marker.element():  # pivot is already sorted
        marker = pivot              # pivot becomes new marker
      else:                         # must relocate pivot
        walk = marker               # find leftmost item greater than value
        while walk != L.first() and L.before(walk).element() > value:
          walk = L.before(walk)
        L.delete(pivot)
        L.add_before(walk, value)   # reinsert value before walk

if __name__ == '__main__':
  L = PositionalList()
  L.add_last(15);L.add_last(22);L.add_last(25);L.add_last(29);L.add_last(36)
  L.add_last(23);L.add_last(53);L.add_last(11);L.add_last(42)
  print(' '.join(str(x) for x in L))
  insertion_sort(L)
  print(' '.join(str(x) for x in L))
```

Python 3.4.3 Shell

File Edit Shell Debug Options Window Help

```
>>>
15 22 25 29 36 23 53 11 42
11 15 22 23 25 29 36 42 53
>>> 
```

Ln: 7 Col: 4

7.6 Case Study: Maintaining Access Frequencies

- The positional list ADT is useful in a number of settings
- maintaining a collection of elements while keeping track of the number of times each element is accessed.
- Keeping such access counts allows us to know which elements are among the most popular
  - browser that keeps track of a user's most accessed URLs,
  - or a music collection that maintains a list of the most frequently played songs for a user.

favorites list ADT

- **access(e):** Access the element e, incrementing its access count, and adding it to the favorites list if it is not already present.
- **remove(e):** Remove element e from the favorites list, if present.
- **top(k):** Return an iteration of the k most accessed elements.

favorites_list.py - C:\course\cs202\2015\ref\Codes\ch07\favorites_list.py (3.4.3)

File Edit Format Run Options Window Help

```python
from positional_list import PositionalList

class FavoritesList:
  """List of elements ordered from most frequently accessed to least."""

  #------------------------------ nested _Item class ------------------------------
  class _Item:
    __slots__ = '_value', '_count'            # streamline memory usage
    def __init__(self, e):
      self._value = e                         # the user's element
      self._count = 0                         # access count initially zero

  #------------------------------- nonpublic utilities -------------------------------
  def _find_position(self, e):
    """Search for element e and return its Position (or None if not found)."""
    walk = self._data.first()
    while walk is not None and walk.element()._value != e:
      walk = self._data.after(walk)
    return walk

  def _move_up(self, p):
    """Move item at Position p earlier in the list based on access count."""
    if p != self._data.first():                     # consider moving...
      cnt = p.element()._count
      walk = self._data.before(p)
      if cnt > walk.element()._count:               # must shift forward
        while (walk != self._data.first() and
               cnt > self._data.before(walk).element()._count):
          walk = self._data.before(walk)
        self._data.add_before(walk, self._data.delete(p))   # delete/reinsert
```

Ln: 34 Col: 27

the composition pattern define a single object that is composed of two or more other objects.

favorites_list.py - C:\course\cs202\2015\ref\Codes\ch07\favorites_list.py (3.4.3)

File Edit Format Run Options Window Help

```
  #------------------------------- pub
  def __init__(self):
    """Create an empty list of favorit
    self._data = PositionalList()

  def __len__(self):
    """Return number of entries on fav
    return len(self._data)

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

  def access(self, e):
    """Access element e, thereby increasing its access count."""
    p = self._find_position(e)                # try to locate existing element
    if p is None:
      p = self._data.add_last(self._Item(e))  # if new, place at end
    p.element()._count += 1                   # always increment count
    self._move_up(p)                          # consider moving forward

  def remove(self, e):
    """Remove element e from the list of favorites."""
    p = self._find_position(e)                # try to locate existing element
    if p is not None:
      self._data.delete(p)                    # delete, if found

  def top(self, k):
    """Generate sequence of top k elements in terms of access count."""
    if not 1 <= k <= len(self):
      raise ValueError('Illegal value for k')
    walk = self._data.first()
    for j in range(k):
      item = walk.element()                   # element of list is _Item
      #yield item._value                       # report user's element
      yield item
      walk = self._data.after(walk)

  def __repr__(self):
    """Create string representation of the favorites list."""
    return ', '.join('({0}:{1})'.format(i._value, i._count) for i in self._data)

if __name__ == '__main__':
  fav = FavoritesList()
  for c in 'hello. this is a test of mtf':       # well, not the mtf part...
    fav.access(c)
    #k = min(5, len(fav))
    #print('Top {0}) {1:25} {2}'.format(k, [x for x in fav.top(k)], fav))

  k = min(5, len(fav))
  print(fav)
  print(' '.join( x._value for x in fav.top(k)))
  print(', '.join('({0}:{1})'.format(i._value, i._count) for i in fav.top(k)))
```

Ln: 29 Col: 40

Python 3.4.3 Shell

File Edit Shell Debug Options Window Help

```
>>>
( :6), (t:4), (s:3), (l:2), (h:2), (i:2), (e:2), (o:2), (f:2), (.:1), (a:1), (m:1)
 t s l h
( :6), (t:4), (s:3), (l:2), (h:2)
>>>
```

Ln: 8 Col: 4

7.6.2 Using a List with the Move-to-Front Heuristic

- if e is the k^{th} most popular element in the favorites list, then accessing it takes O(k) time.
- **locality of reference**: once an element is accessed it is more likely to be accessed again in the near future.
- A ***heuristic***, or rule of thumb, that attempts to take advantage of the locality of reference that is present in an access sequence is the ***move-to-front heuristic***.

Performance: n elements and the following series of n^2 accesses

- element 1 is accessed n times
- element 2 is accessed n times
- $\cdots$
- element n is accessed n times.

If we store the elements sorted by their access counts, inserting each element the first time it is accessed, then

- each access to element 1 runs in $O(1)$ time
- each access to element 2 runs in $O(2)$ time
- $\cdots$
- each access to element n runs in $O(n)$ time.

Thus, the total time for performing the series of accesses is proportional to

$$n+2n+3n+\cdots+n\cdot n = n(1+2+3+\cdots+n) = n\cdot\frac{n(n+1)}{2},$$

which is $O(n^3)$.

On the other hand, if we use the move-to-front heuristic, inserting each element the first time it is accessed, then

- each subsequent access to element 1 takes $O(1)$ time
- each subsequent access to element 2 takes $O(1)$ time
- $\cdots$
- each subsequent access to element n runs in $O(1)$ time.

So the running time for performing all the accesses in this case is $O(n^2)$. Thus,

The Trade-Offs with the Move-to-Front Heuristic

If we no longer maintain the elements of the favorites list ordered by their access counts, when we are asked to find the k most accessed elements, we need to search for them. We will implement the top(k) method as follows:

1. We copy all entries of our favorites list into another list, named temp.
2. We scan the temp list k times. In each scan, we find the entry with the largest access count, remove this entry from temp, and report it in the results.

This implementation of method top takes $O(kn)$ time. Thus, when k is a constant, method top runs in $O(n)$ time. This occurs, for example, when we want to get the "top ten" list. However, if k is proportional to n, then top runs in $O(n^2)$ time. This occurs, for example, when we want a "top 25%" list.

In Chapter 9 we will introduce a data structure that will allow us to implement top in $O(n + k \log n)$ time (see Exercise P-9.54), and more advanced techniques could be used to perform top in $O(n + k \log k)$ time.

We could easily achieve $O(n \log n)$ time if we use a standard sorting algorithm to reorder the temporary list before reporting the top k (see Chapter 12); this approach would be preferred to the original in the case that k is $\Omega(\log n)$. (Recall the big-Omega notation introduced in Section 3.3.1 to give an asymptotic lower bound on the running time of an algorithm.) There is a more specialized sorting algorithm (see Section 12.4.2) that can take advantage of the fact that access counts are integers in order to achieve $O(n)$ time for top, for any value of k.

favorites_list_mtf.py - C:\course\cs202\2015\ref\Codes\ch07\tmp\favorites_list_mtf.py (3.4.3)

File Edit Format Run Options Window Help

```python
from favorites_list import FavoritesList
from positional_list import PositionalList

class FavoritesListMTF(FavoritesList):
  """List of elements ordered with move-to-front heuristic."""

  # we override _move_up to provide move-to-front semantics
  def _move_up(self, p):
    """Move accessed item at Position p to front of list."""
    if p != self._data.first():
      self._data.add_first(self._data.delete(p))        # delete/reinsert

  # we override top because list is no longer sorted
  def top(self, k):
    """Generate sequence of top k elements in terms of access count."""
    if not 1 <= k <= len(self):
      raise ValueError('Illegal value for k')

    # we begin by making a copy of the original list
    temp = PositionalList()
    for item in self._data:              # positional lists support iteration
      temp.add_last(item)

    # we repeatedly find, report, and remove element with largest count
    for j in range(k):
      # find and report next highest from temp
      highPos = temp.first()
      walk = temp.after(highPos)
      while walk is not None:
        if walk.element()._count > highPos.element()._count:
          highPos = walk
        walk = temp.after(walk)
      # we have found the element with highest count
      #yield highPos.element()._value                 # report element to user
      yield highPos.element()
      temp.delete(highPos)                            # remove from temp list

if __name__ == '__main__':
  fav = FavoritesListMTF()
  for c in 'hello. this is a test of mtf':
    fav.access(c)
    #k = min(5, len(fav))
    #print('Top {0}) {1:25} {2}'.format(k, [x for x in fav.top(k)], fav))

  k = min(5, len(fav))
  print(', '.join('({0}:{1})'.format(i._value, i._count) for i in fav.top(k)))
```

Ln: 35 Col: 29

Python 3.4.3 Shell

File Edit Shell Debug Options Window Help

```
>>>
( :6), (t:4), (s:3), (f:2), (o:2)
>>>
```

Ln: 6 Col: 4

live.gnome.org/**Dia**

- draw entity relationship diagrams, Unified Modeling Language diagrams, flowcharts, network diagrams, and many other diagrams.
- The UML models not only application structure, behavior, and architecture, but also business process and data structure.
- http://net.pku.edu.cn/~course/cs202/2015/resource/software/dia-setup-0.97.2-2-unsigned.exe

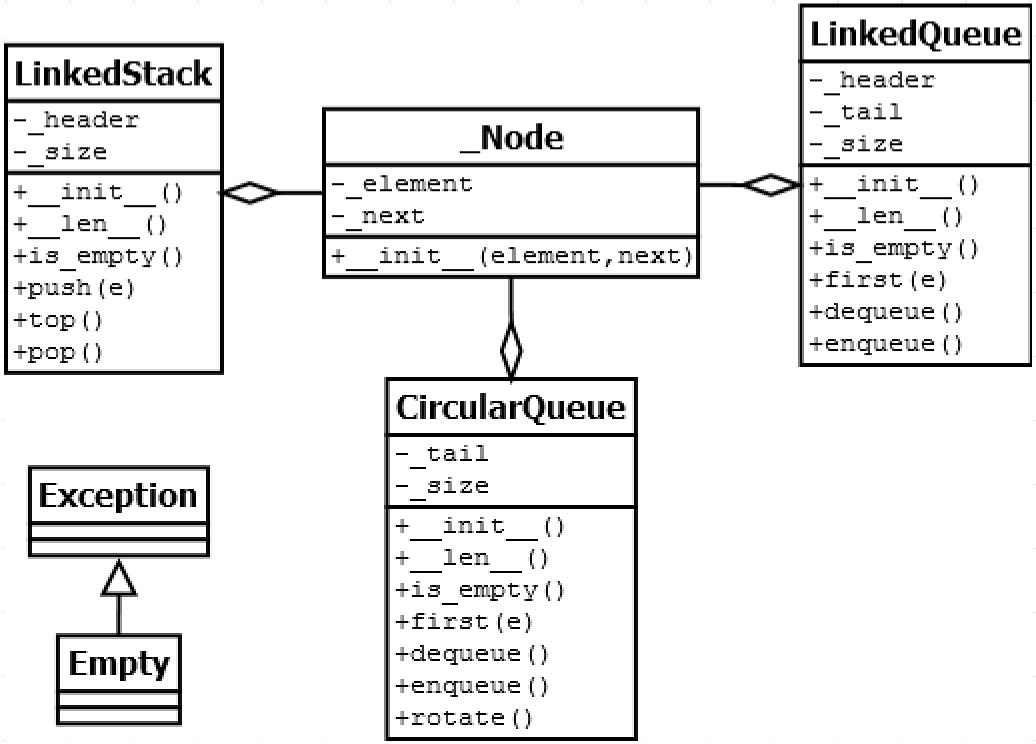
Legend

- --->依赖(dependency)：表示一个类依赖于另一个类的定义。类A使用到了另一个类B。
- ——◇聚集(aggregation)：体现的是整体与部分、拥有的关系，即has-a的关系
- ——▷继承/泛化(generalization)：指的是一个类（称为子类）继承另外的一个类（称为父类）的功能，并可以增加它自己的新功能的能力。
  - 继承是类与类之间最常见的关系，可分单重继承，多重继承

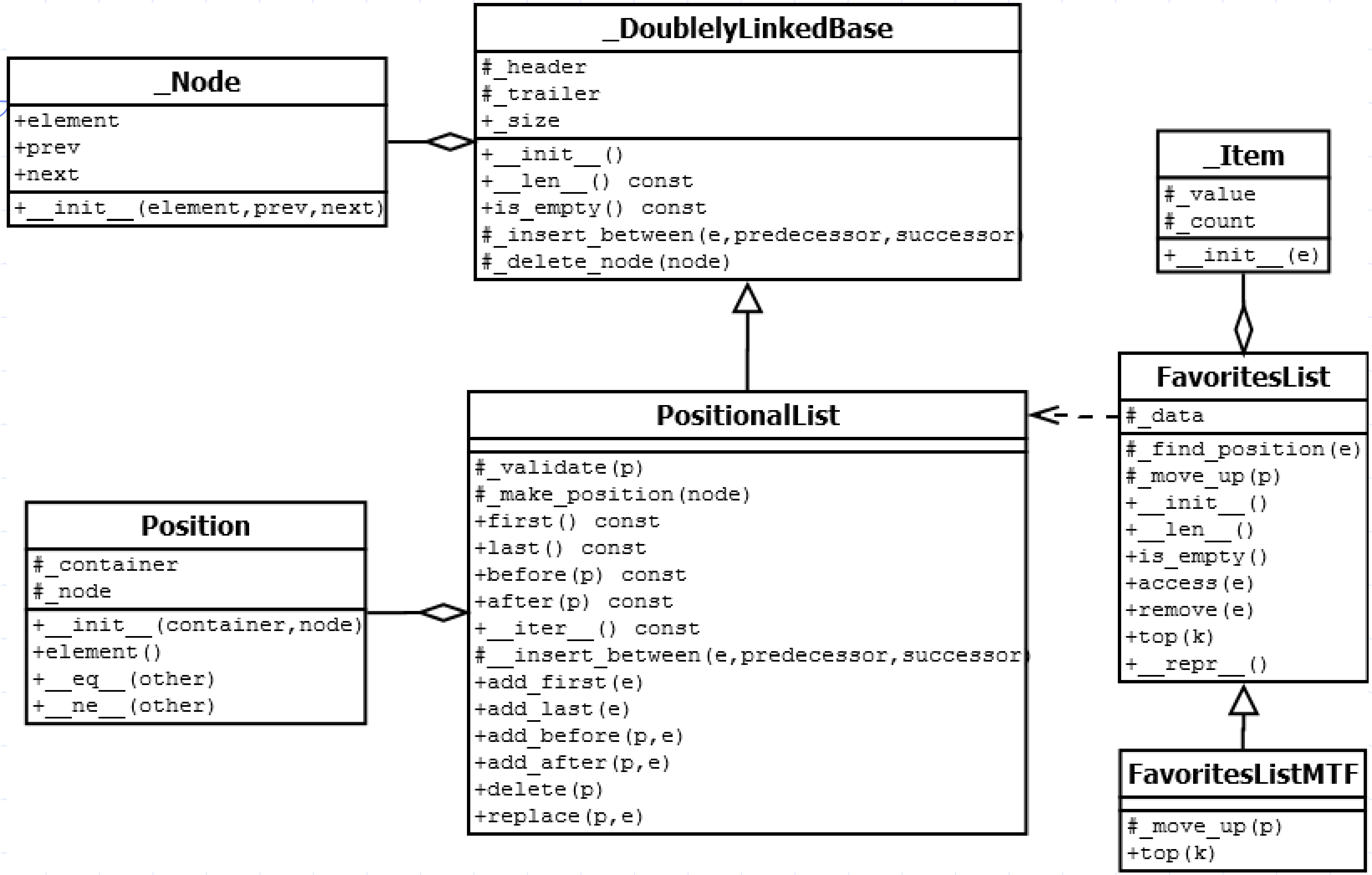
python 类中下划线的作用

- _xxx 不能用于'from module import *'
  - 以单下划线开头的表示的是protected类型的变量。
  - 即保护类型只能允许其本身与子类进行访问。
- __xxx 双下划线的表示的是私有类型的变量
  - 只能是允许这个类本身进行访问了。连子类也不可以
- __xxx___ 定义的是特列方法。
  - 像__init__之类的

Summary: Linked Lists



Summary: Double Linked Lists



7.7 Link-Based vs. Array-Based Sequences

- The dichotomy between these approaches presents a common design decision when choosing an appropriate implementation of a data structure.
- There is not a one-size-fits-all solution, as each offers distinct advantages and disadvantages.

Advantages of Array-Based Sequences (1/2)

- *Arrays provide $O(1)$-time access to an element based on an integer index.* The ability to access the k^{th} element for any k in $O(1)$ time is a hallmark advantage of arrays (see Section 5.2). In contrast, locating the k^{th} element in a linked list requires $O(k)$ time to traverse the list from the beginning, or possibly $O(n-k)$ time, if traversing backward from the end of a doubly linked list.

- *Operations with equivalent asymptotic bounds typically run a constant factor more efficiently with an array-based structure versus a linked structure.* As an example, consider the typical enqueue operation for a queue. Ignoring the issue of resizing an array, this operation for the ArrayQueue class (see Code Fragment 6.7) involves an arithmetic calculation of the new index, an increment of an integer, and storing a reference to the element in the array. In contrast, the process for a LinkedQueue (see Code Fragment 7.8) requires the instantiation of a node, appropriate linking of nodes, and an increment of an integer. While this operation completes in $O(1)$ time in either model, the actual number of CPU operations will be more in the linked version, especially given the instantiation of the new node.

Advantages of Array-Based Sequences (2/2)

- *Array-based representations typically use proportionally less memory than linked structures.* This advantage may seem counterintuitive, especially given that the length of a dynamic array may be longer than the number of elements that it stores. Both array-based lists and linked lists are referential structures, so the primary memory for storing the actual objects that are elements is the same for either structure. What differs is the auxiliary amounts of memory that are used by the two structures. For an array-based container of n elements, a typical worst case may be that a recently resized dynamic array has allocated memory for $2n$ object references. With linked lists, memory must be devoted not only to store a reference to each contained object, but also explicit references that link the nodes. So a singly linked list of length n already requires $2n$ references (an element reference and next reference for each node). With a doubly linked list, there are $3n$ references.

Advantages of Link-Based Sequences (1/2)

- *Link-based structures provide worst-case time bounds for their operations.* This is in contrast to the amortized bounds associated with the expansion or contraction of a dynamic array (see Section 5.3).

  When many individual operations are part of a larger computation, and we only care about the total time of that computation, an amortized bound is as good as a worst-case bound precisely because it gives a guarantee on the sum of the time spent on the individual operations.

  However, if data structure operations are used in a real-time system that is designed to provide more immediate responses (e.g., an operating system, Web server, air traffic control system), a long delay caused by a single (amortized) operation may have an adverse effect.

Advantages of Link-Based Sequences (2/2)

- *Link-based structures support $O(1)$-time insertions and deletions at arbitrary positions.* The ability to perform a constant-time insertion or deletion with the PositionalList class, by using a Position to efficiently describe the location of the operation, is perhaps the most significant advantage of the linked list.

  This is in stark contrast to an array-based sequence. Ignoring the issue of resizing an array, inserting or deleting an element from the end of an array-based list can be done in constant time. However, more general insertions and deletions are expensive. For example, with Python's array-based list class, a call to insert or pop with index k uses $O(n-k+1)$ time because of the loop to shift all subsequent elements (see Section 5.4).