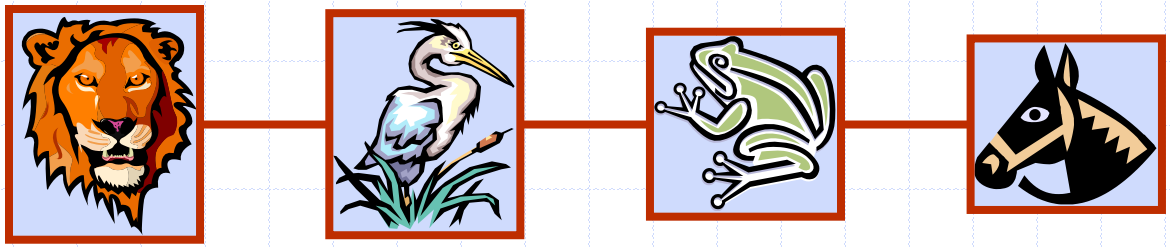
Linked Lists



Recap: Array-based Stack in Python

```
1  class ArrayStack:
2    """LIFO Stack implementation using a Python list as underlying storage."""
3
4    def __init__(self):
5      """Create an empty stack."""
6      self._data = [ ]                       # nonpublic list instance
7
8    def __len__(self):
9      """Return the number of elements in the stack."""
10     return len(self._data)
11
12   def is_empty(self):
13     """Return True if the stack is empty."""
14     return len(self._data) == 0
15
16   def push(self, e):
17     """Add element e to the top of the stack."""
18     self._data.append(e)                   # new item stored at end of list
19
```

```
20   def top(self):
21     """Return (but do not remove) the element at the top of the stack.
22
23     Raise Empty exception if the stack is empty.
24     """
25     if self.is_empty( ):
26       raise Empty('Stack is empty')
27     return self._data[-1]                  # the last item in the list
28
29   def pop(self):
30     """Remove and return the element from the top of the stack (i.e., LIFO).
31
32     Raise Empty exception if the stack is empty.
33     """
34     if self.is_empty( ):
35       raise Empty('Stack is empty')
36     return self._data.pop( )               # remove last item from list
```

Recap: Queue in Python (1/2)

```
1  class ArrayQueue:
2    """FIFO queue implementation using a Python list as underlying storage."""
3    DEFAULT_CAPACITY = 10          # moderate capacity for all new queues
4
5    def __init__(self):
6      """Create an empty queue."""
7      self._data = [None] * ArrayQueue.DEFAULT_CAPACITY
8      self._size = 0
9      self._front = 0
10
11   def __len__(self):
12     """Return the number of elements in the queue."""
13     return self._size
14
15   def is_empty(self):
16     """Return True if the queue is empty."""
17     return self._size == 0
18
```

```
19   def first(self):
20     """Return (but do not remove) the element at the front of the queue.
21
22     Raise Empty exception if the queue is empty.
23     """
24     if self.is_empty():
25       raise Empty('Queue is empty')
26     return self._data[self._front]
27
28   def dequeue(self):
29     """Remove and return the first element of the queue (i.e., FIFO).
30
31     Raise Empty exception if the queue is empty.
32     """
33     if self.is_empty():
34       raise Empty('Queue is empty')
35     answer = self._data[self._front]
36     self._data[self._front] = None          # help garbage collection
37     self._front = (self._front + 1) % len(self._data)
38     self._size -= 1
39     return answer
```

Recap: Queue in Python (2/2)

```
40    def enqueue(self, e):
41      """Add an element to the back of queue."""
42      if self._size == len(self._data):
43        self._resize(2 * len(self.data))          # double the array size
44      avail = (self._front + self._size) % len(self._data)
45      self._data[avail] = e
46      self._size += 1
47
48    def _resize(self, cap):                       # we assume cap >= len(self)
49      """Resize to a new list of capacity >= len(self)."""
50      old = self._data                            # keep track of existing list
51      self._data = [None] * cap                   # allocate list with new capacity
52      walk = self._front
53      for k in range(self._size):                 # only consider existing elements
54        self._data[k] = old[walk]                 # intentionally shift indices
55        walk = (1 + walk) % len(old)              # use old size as modulus
56      self._front = 0                             # front has been realigned
```

Assignment #5: Stacks and Queues

- R-6.5 Implement a function that reverses a list of elements by pushing them onto a stack in one order, and writing them back to the list in reversed order.
- R-6.13 Suppose you have a deque D containing the numbers (1,2,3,4,5,6,7,8), in this order. Suppose further that you have an initially empty queue Q. Give a code fragment that uses only D and Q (and no other variables) and results in D storing the elements in the order (1,2,3,5,4,6,7,8).
- R-6.14 Repeat the previous problem using the deque D and an initially empty stack S.
- Find spans in an array. Given an array arr[], the SPAN s[i] of arr[i] is the maximum number of consecutive elements arr[j] immediately before arr[i] such that arr[j] <= arr[i]. (C++ codes: spans1.cpp, spans2.cpp). Implement both the spans1 which runs in $O(n^2)$ time and the spans2 which runs in O(n) time in Python.

R6.5 reverses a list of elements by pushing them onto a stack in one order, and writing them back to the list in reversed order.

R-6.5.py - C:\course\cs202\2015\ref\Codes\ch06\R-6.5.py (3.4.3)

File Edit Format Run Options Window Help

```
from array_stack import ArrayStack

def reverse(sequence):
    """A nonrecursive method for reversing a Python list."""
    S = ArrayStack()
    for j in range(len(sequence)):
        S.push(sequence[j])
    for j in range(len(sequence)):
        sequence[j] = S.pop( )

if __name__ == '__main__':
    sequence = list("ABCDEFG")
    print(sequence)
    reverse(sequence)
    print(sequence)
```

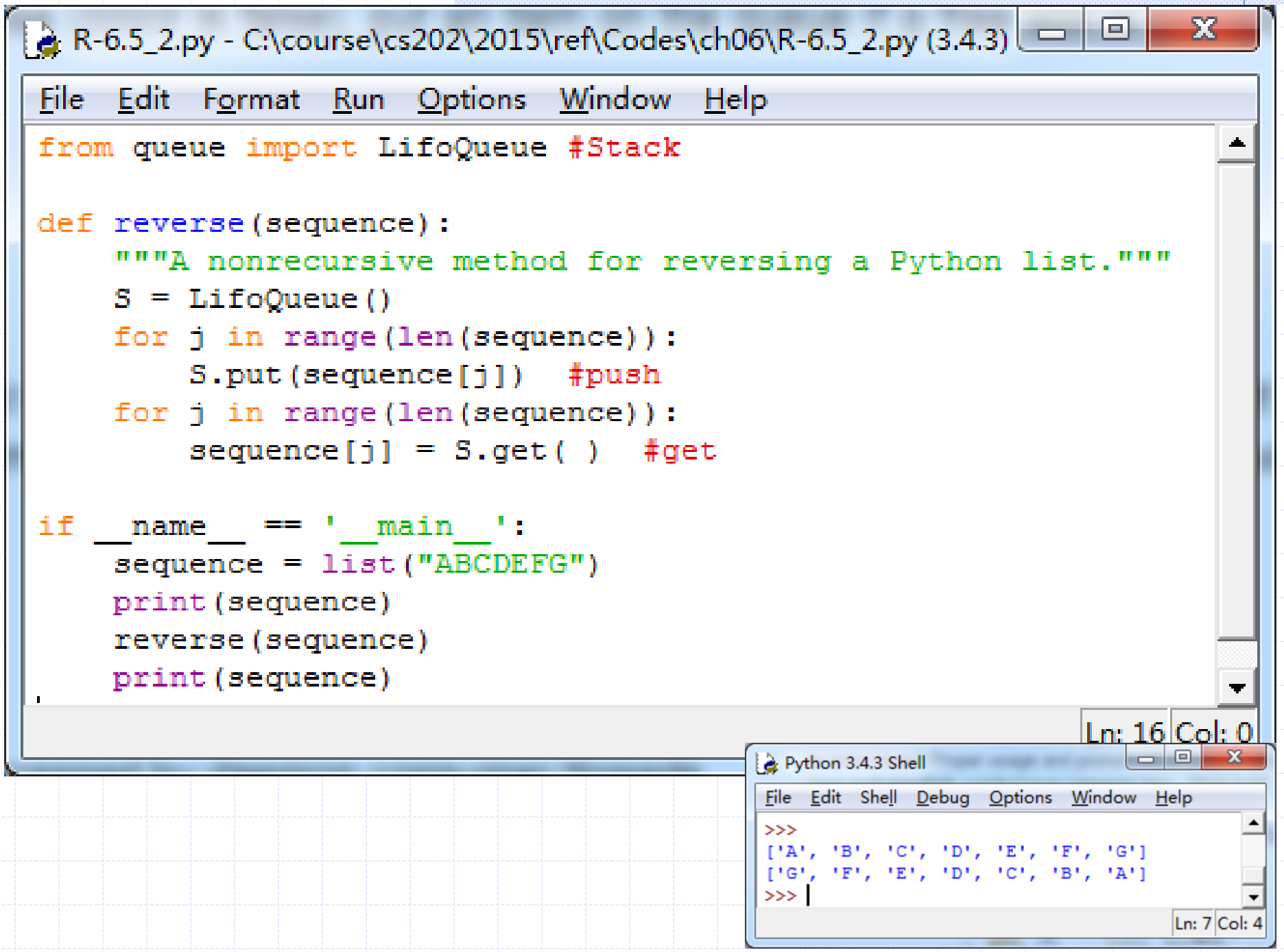
Ln: 16 Col: 0

Python 3.4.3 Shell

File Edit Shell Debug Options Window Help

```
>>>
['A', 'B', 'C', 'D', 'E', 'F', 'G']
['G', 'F', 'E', 'D', 'C', 'B', 'A']
>>>
```

Ln: 7 Col: 4


R-6.5_2.py - C:\course\cs202\2015\ref\Codes\ch06\R-6.5_2.py (3.4.3)
File Edit Format Run Options Window Help
from queue import LifoQueue #Stack
def reverse(sequence):
"""A nonrecursive method for reversing a Python list."""
S = LifoQueue()
for j in range(len(sequence)):
S.put(sequence[j]) #push
for j in range(len(sequence)):
sequence[j] = S.get( ) #get
if __name__ == '__main__':
sequence = list("ABCDEFG")
print(sequence)
reverse(sequence)
print(sequence)
Ln: 16 Col: 0
Python 3.4.3 Shell
File Edit Shell Debug Options Window Help
>>>
['A', 'B', 'C', 'D', 'E', 'F', 'G']
['G', 'F', 'E', 'D', 'C', 'B', 'A']
>>>
Ln: 7 Col: 4

R-6.13 Suppose you have a deque D containing the numbers (1,2,3,4,5,6,7,8), in this order. Suppose further that you have an initially empty queue Q. Give a code fragment that uses only D and Q (and no other variables) and results in D storing the elements in the order (1,2,3,5,4,6,7,8).

R-6.13.py - C:\course\cs202\2015\ref\Codes\ch06\R-6.13.py (3.4...

File Edit Format Run Options Window Help

```
from collections import deque
from queue import Queue

d = deque()
d.extend(x for x in range(1,9))

q = Queue()
q.put(d.popleft());q.put(d.popleft());q.put(d.popleft())

d.append(d.popleft())    # 4 appended right
q.put(d.popleft())
q.put(d.pop())

q.put(d.popleft());q.put(d.popleft());q.put(d.popleft())


l = []
l = [item for item in q.queue] #dumping queue into list
'''
while q.qsize() > 0:
    l.append(q.get())
'''
print(l)
```

Ln: 24 Col: 0

Python 3.4.3 Shell

File Edit Shell Debug Options Window Help

```
>>>
[1, 2, 3, 5, 4, 6, 7, 8]
>>>
```

Ln: 6 Col: 4

R-6.14 Repeat the previous problem using the deque D and an initially empty stack S.

R-6.14.py - C:\course\cs202\2015\ref\Codes\ch06\R-6.14.py (3.4...

File Edit Format Run Options Window Help

```
from collections import deque
from queue import LifoQueue #Stack

d = deque()
d.extend(x for x in range(1,9))

S = LifoQueue()
print(d.popleft());print(d.popleft());print(d.popleft())

S.put(d.popleft())    # 4
print(d.popleft())
print(S.get())

print(d.popleft());print(d.popleft());print(d.popleft())
```

Ln: 15 Col: 0

Python 3.4.3 Shell

File Edit Shell Debug Options Window Help

```
>>>
1
2
3
5
4
6
7
8
>>>
```

Ln: 13 Col: 4

Find spans in an array

span1.py - C:\course\cs202\2015\ref\Codes\ch06\span1.py (3.4.3)

File Edit Format Run Options Window Help

```
#find spans in an array. O(n^2)

arr = [6,3,4,5,2]
new_arr = [0]*len(arr)

for i in range(len(arr)):
    j = i - 1
    count = 1
    while arr[j] <= arr[i] and j>=0:
        count += 1
        j -= 1

    new_arr[i] = count

print (str(arr).strip())
print (str(new_arr).strip())
```

Ln: 16 Col: 0

Python 3.4.3 Shell

File Edit Shell Debug Options Window Help

```
>>>
[6, 3, 4, 5, 2]
[1, 1, 2, 3, 1]
>>>
```

Ln: 14 Col: 4

Find spans in an array

span2.py - C:\course\cs202\2015\ref\Codes\ch06\span2.py (3.4.3)

File Edit Format Run Options Window Help

```
#find spans in an array. O(n)
from queue import LifoQueue #Stack

arr = [6,3,4,5,2]
new_arr = [0]*len(arr)

S = LifoQueue()
for i in range(len(arr)):
    while S.qsize()>0 :
        pivot = S.get()              #no top() in Stack, have to push again
        if arr[pivot] > arr[i]:
            S.put(pivot)
            break

    if S.qsize() == 0:
        new_arr[i] = i + 1;
    else:
        new_arr[i] = i - pivot

    S.put(i)          #push

print (str(arr).strip())
print (str(new_arr).strip())
```

Python 3.4.3 Shell

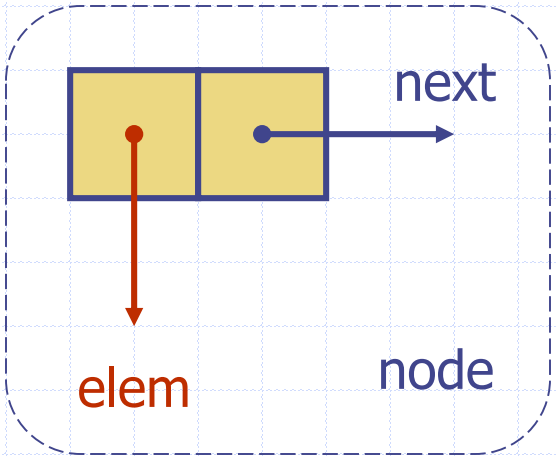
File Edit Shell Debug Options Window Help

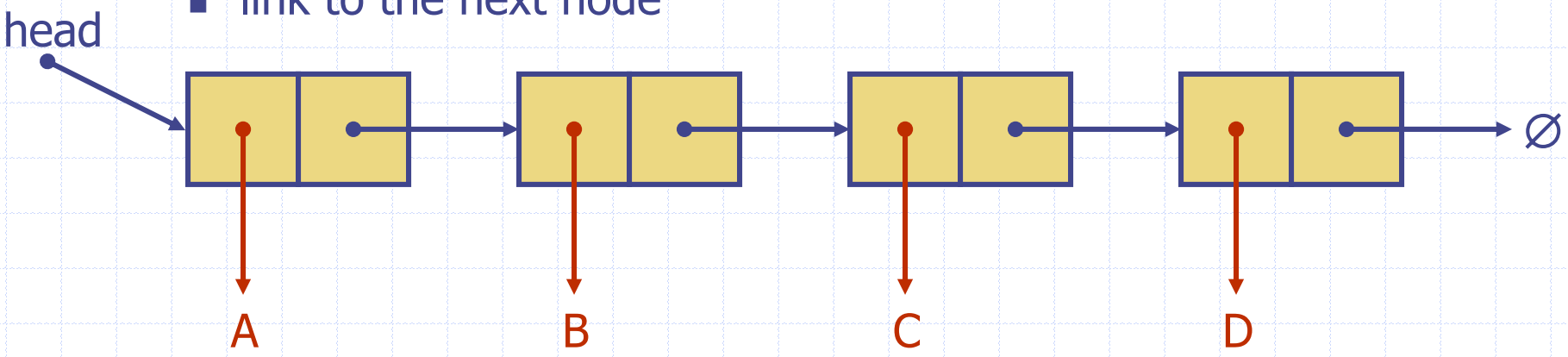
```
>>>
[6, 3, 4, 5, 2]
[1, 1, 2, 3, 1]
>>>
```

Ln: 14 Col: 4

Singly Linked List

- A singly linked list is a concrete data structure consisting of a sequence of nodes, starting from a head pointer
- Each node stores
  - element
  - link to the next node





The Node Class for List Nodes

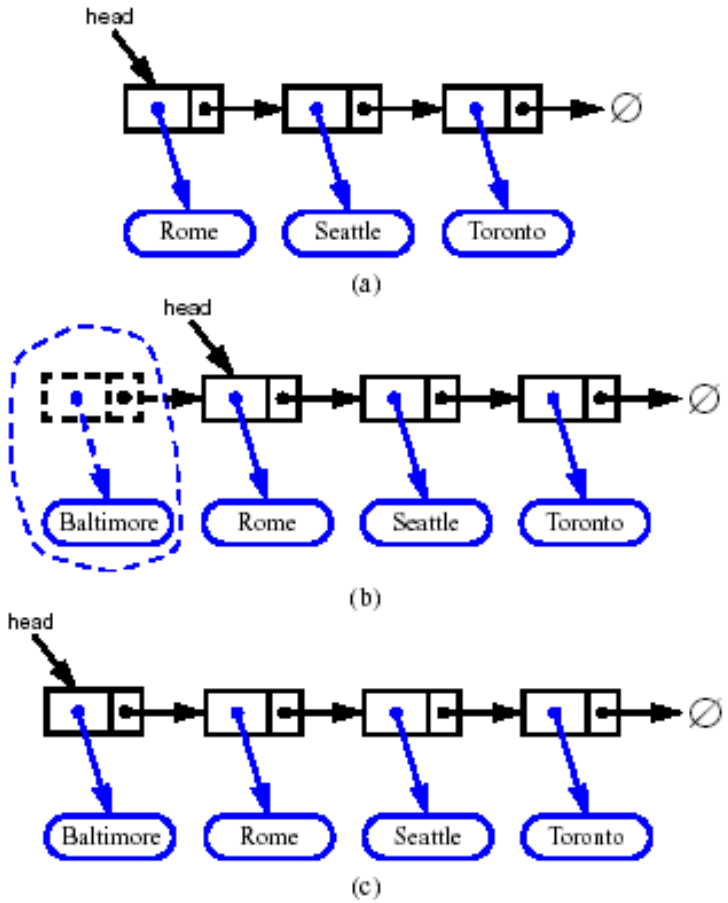
```
class _Node:
  """Lightweight, nonpublic class for storing a singly linked node."""
  __slots__ = '_element', '_next'          # streamline memory usage

  def __init__(self, element, next):       # initialize node's fields
    self._element = element                # reference to user's element
    self._next = next                      # reference to next node
```

Code Fragment 7.4: A lightweight _Node class for a singly linked list.

By default, Python represents each namespace with an instance of the built-in dict class (see Section 1.2.3) that maps identifying names in that scope to the associated objects.
provide a class-level member named slots that is assigned to a fixed sequence of strings that serve as names
for instance variables.

Inserting at the Head

1. Allocate a new node
2. Insert new element
3. Have new node point to old head
4. Update head to point to new node



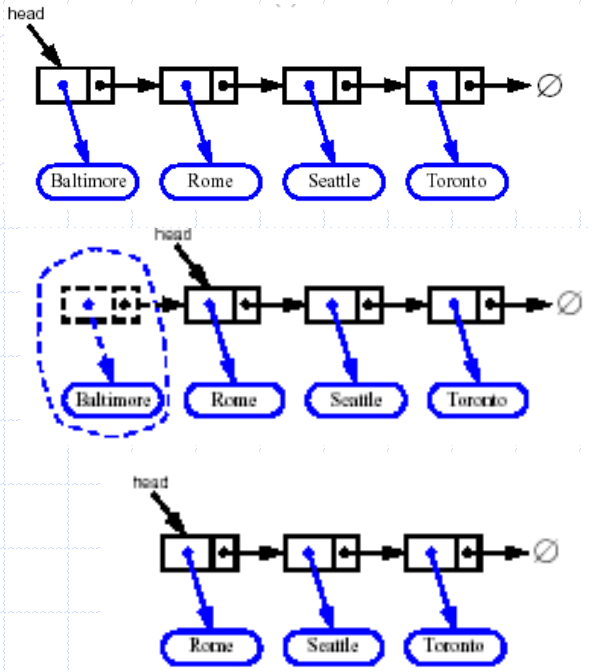
Algorithm add_first(L, e):

```
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}
```

Code Fragment 7.1: Inserting a new element at the beginning of a singly linked list L. 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.

Removing at the Head

1. Update head to point to next node in the list
2. Allow garbage collector to reclaim the former first node



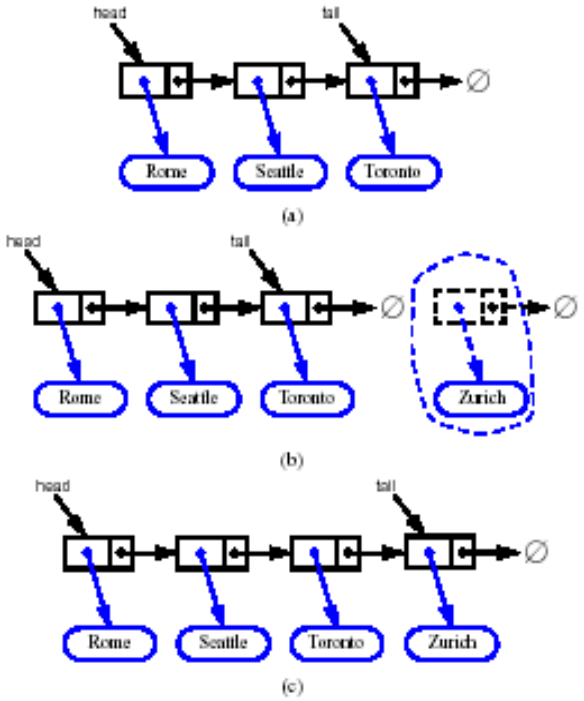
```
Algorithm remove_first(L):
    if L.head is None then
        Indicate an error: the list is empty.
    L.head = L.head.next          {make head point to next node (or None)}
    L.size = L.size − 1           {decrement the node count}
```

Code Fragment 7.3: Removing the node at the beginning of a singly linked list.

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. 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. If we want to support such an operation efficiently, we will need to make our list ***doubly linked*** (as we do in Section 7.3).

Inserting at the Tail

1. Allocate a new node
2. Insert new element
3. Have new node point to null
4. Have old last node point to new node
5. Update tail to point to new node



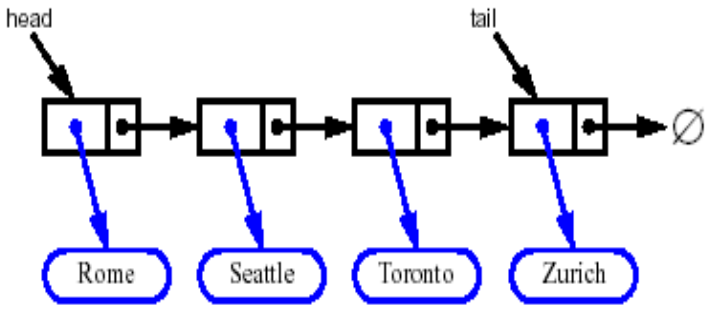
Algorithm add_last(L, e):

```
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}
```

Code Fragment 7.2: Inserting a new node at the end of a singly linked list. 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.

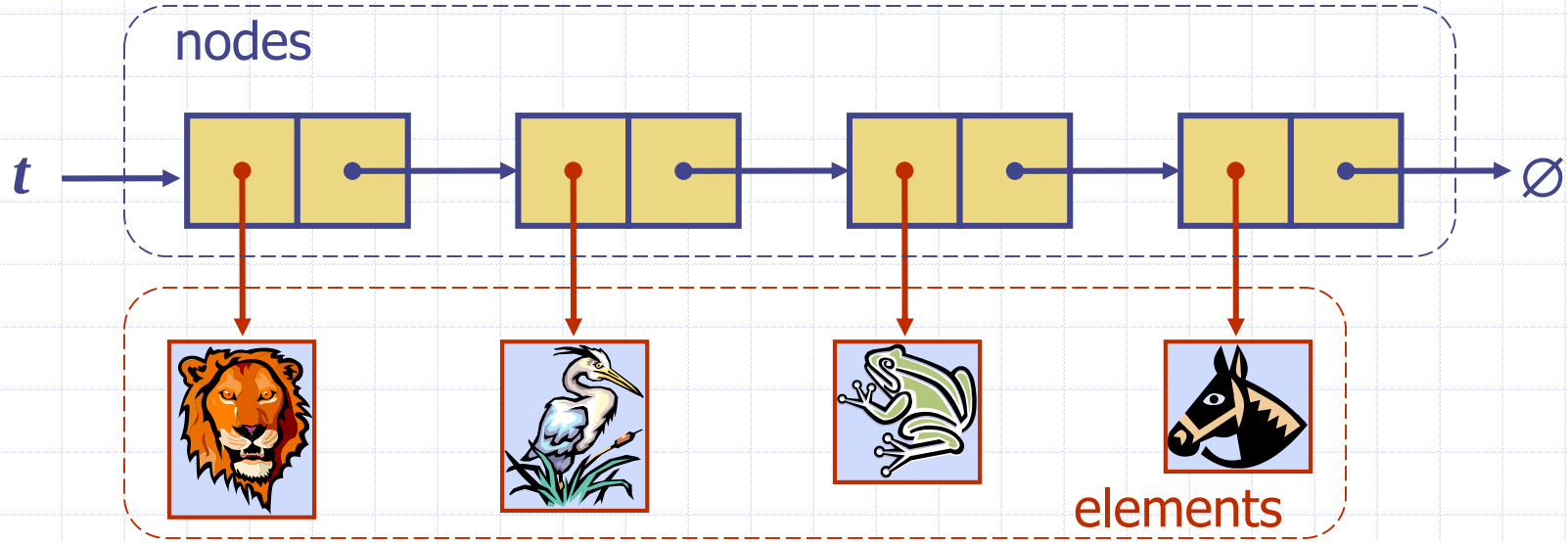
Removing at the Tail

- Removing at the tail of a singly linked list is not efficient!
- There is no constant-time way to update the tail to point to the previous node



Stack as a Linked List

- We can implement a stack with a singly linked list
- The top element is stored at the first node of the list
- The space used is $O(n)$ and each operation of the Stack ADT takes $O(1)$ time



Linked-List Stack in Python

```
1  class LinkedStack:
2    """LIFO Stack implementation using a singly linked list for storage."""
3
4    #---------------------------- nested _Node class ----------------------------
5    class _Node:
6      """Lightweight, nonpublic class for storing a singly linked node."""
7      __slots__ = '_element', '_next'         # streamline memory usage
8
9      def __init__(self, element, next):      # initialize node's fields
10       self._element = element               # reference to user's element
11       self._next = next                     # reference to next node
12
13   #------------------------------ stack methods ------------------------------
14   def __init__(self):
15     """Create an empty stack."""
16     self._head = None                       # reference to the head node
17     self._size = 0                          # number of stack elements
18
19   def __len__(self):
20     """Return the number of elements in the stack."""
21     return self._size
22
```

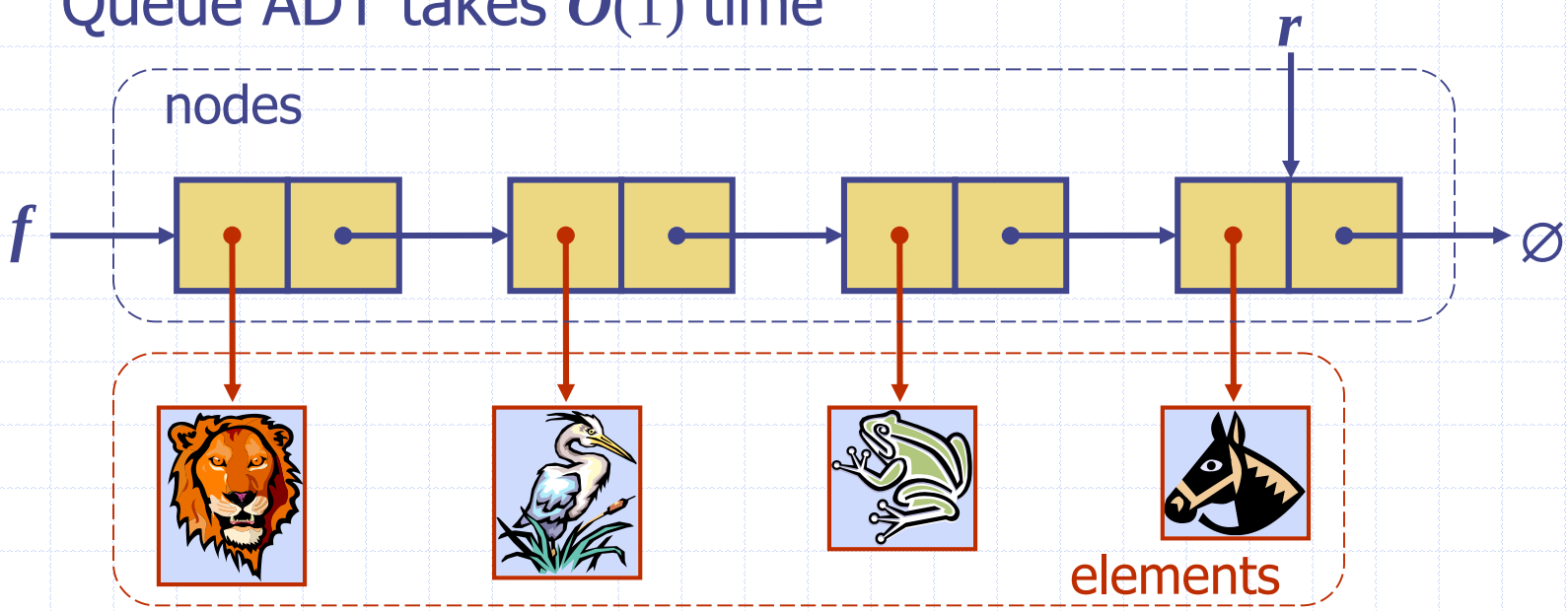
```
23   def is_empty(self):
24     """Return True if the stack is empty."""
25     return self._size == 0
26
27   def push(self, e):
28     """Add element e to the top of the stack."""
29     self._head = self._Node(e, self._head)    # create and link a new node
30     self._size += 1
31
32   def top(self):
33     """Return (but do not remove) the element at the top of the stack.
34
35     Raise Empty exception if the stack is empty.
36     """
37     if self.is_empty():
38       raise Empty('Stack is empty')
39     return self._head._element                # top of stack is at head of list
```

```
40   def pop(self):
41     """Remove and return the element from the top of the stack (i.e., LIFO).
42
43     Raise Empty exception if the stack is empty.
44     """
45     if self.is_empty():
46       raise Empty('Stack is empty')
47     answer = self._head._element
48     self._head = self._head._next           # bypass the former top node
49     self._size -= 1
50     return answer
```

Operation	Running Time
S.push(e)	$O(1)$
S.pop( )	$O(1)$
S.top( )	$O(1)$
len(S)	$O(1)$
S.is_empty( )	$O(1)$

Queue as a Linked List

- We can implement a queue with a singly linked list
  - The front element is stored at the first node
  - The rear element is stored at the last node
- The space used is $O(n)$ and each operation of the Queue ADT takes $O(1)$ time



Linked-List Queue in Python

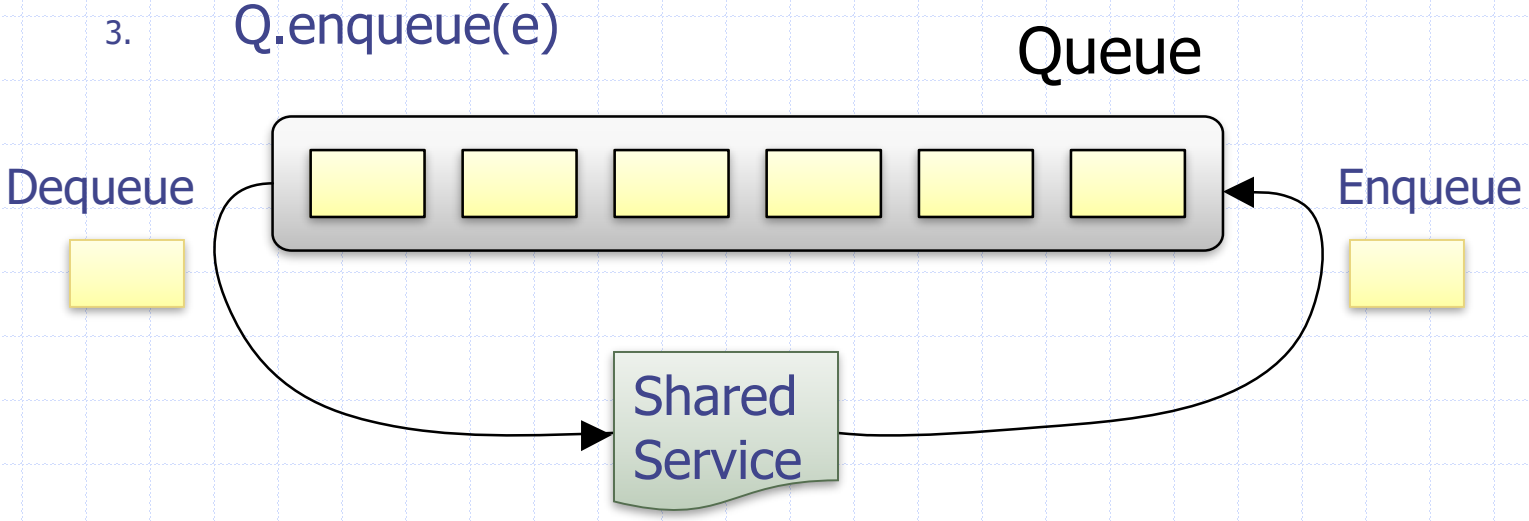
```
1  class LinkedQueue:
2    """FIFO queue implementation using a singly linked list for storage."""
3
4    class _Node:
5      """Lightweight, nonpublic class for storing a singly linked node."""
6      (omitted here; identical to that of LinkedStack._Node)
7
8    def __init__(self):
9      """Create an empty queue."""
10     self._head = None
11     self._tail = None
12     self._size = 0                          # number of queue elements
13
14   def __len__(self):
15     """Return the number of elements in the queue."""
16     return self._size
17
18   def is_empty(self):
19     """Return True if the queue is empty."""
20     return self._size == 0
21
22   def first(self):
23     """Return (but do not remove) the element at the front of the queue."""
24     if self.is_empty():
25       raise Empty('Queue is empty')
26     return self._head._element              # front aligned with head of list
```

```
27   def dequeue(self):
28     """Remove and return the first element of the queue (i.e., FIFO).
29
30     Raise Empty exception if the queue is empty.
31     """
32     if self.is_empty():
33       raise Empty('Queue is empty')
34     answer = self._head._element
35     self._head = self._head._next
36     self._size -= 1
37     if self.is_empty():                     # special case as queue is empty
38       self._tail = None                     # removed head had been the tail
39     return answer
40
41   def enqueue(self, e):
42     """Add an element to the back of queue."""
43     newest = self._Node(e, None)            # node will be new tail node
44     if self.is_empty():
45       self._head = newest                   # special case: previously empty
46     else:
47       self._tail._next = newest
48     self._tail = newest                     # update reference to tail node
49     self._size += 1
```

supporting worst-case $O(1)$-time for all operations

Application: Round Robin Schedulers

- We can implement a **round robin** scheduler using a queue Q by repeatedly performing the following steps:
  1. e = Q.dequeue()
  2. Service element e
  3. Q.enqueue(e)



Queue with a Circularly Linked List in Python

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

```
20   def first(self):
21     """Return (but do not remove) the element at the front of the queue.
22
23     Raise Empty exception if the queue is empty.
24     """
25     if self.is_empty():
26       raise Empty('Queue is empty')
27     head = self._tail._next
28     return head._element
29
30   def dequeue(self):
31     """Remove and return the first element of the queue (i.e., FIFO).
32
33     Raise Empty exception if the queue is empty.
34     """
35     if self.is_empty():
36       raise Empty('Queue is empty')
37     oldhead = self._tail._next
38     if self._size == 1:                  # removing only element
39       self._tail = None                  # queue becomes empty
40     else:
41       self._tail._next = oldhead._next   # bypass the old head
42     self._size -= 1
43     return oldhead._element
44
45   def enqueue(self, e):
46     """Add an element to the back of queue."""
47     newest = self._Node(e, None)         # node will be new tail node
48     if self.is_empty():
49       newest._next = newest              # initialize circularly
50     else:
51       newest._next = self._tail._next    # new node points to head
52       self._tail._next = newest          # old tail points to new node
53     self._tail = newest                  # new node becomes the tail
54     self._size += 1
55
56   def rotate(self):
57     """Rotate front element to the back of the queue."""
58     if self._size > 0:
59       self._tail = self._tail._next      # old head becomes new tail
```