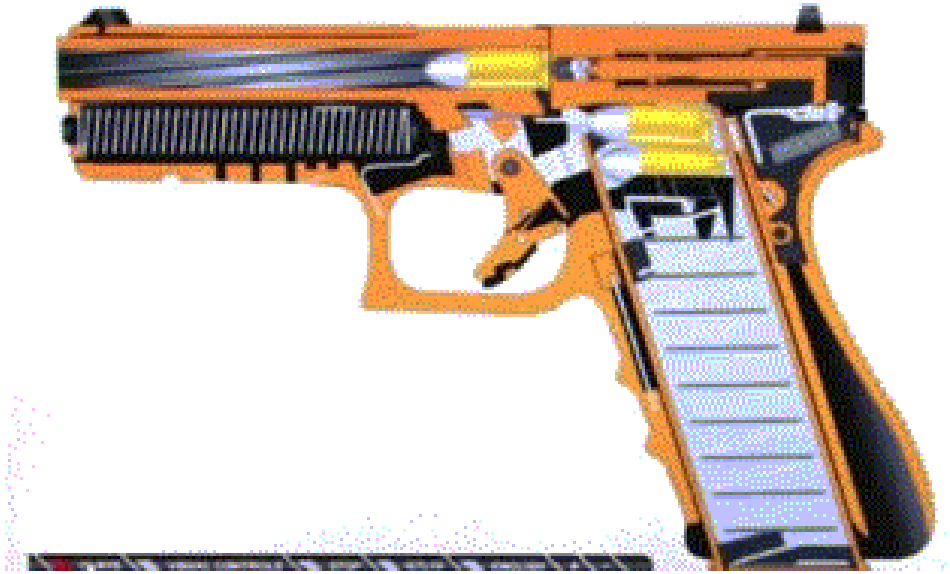
SCNU - UOA
OBJECT-ORIENTED PROGRAMMING

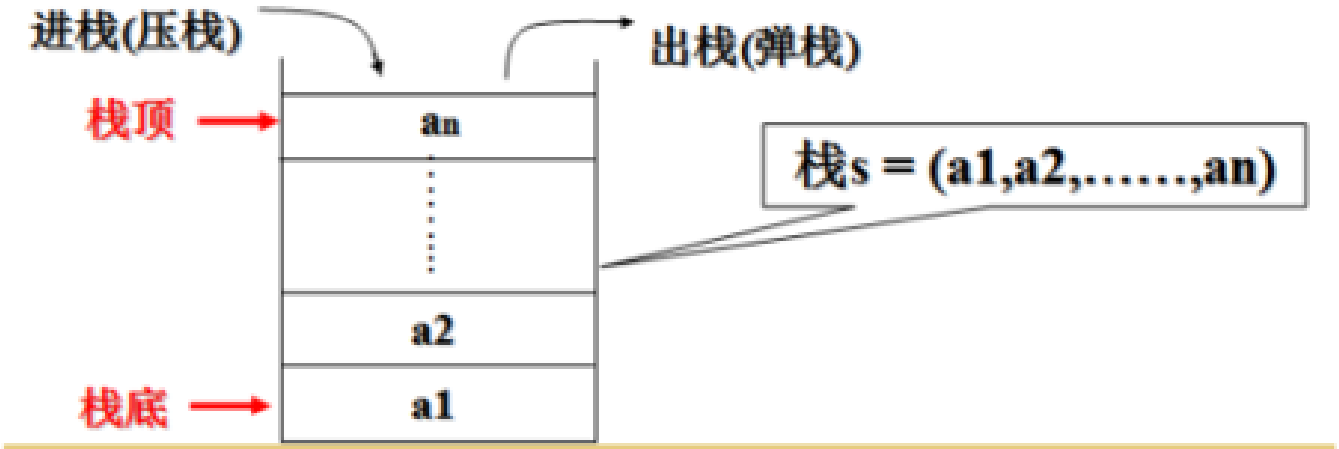
Data Structures and Algorithms

Stack and Queue

3. Stack

- Last-in first-out scheme





2. The Stack ADT #堆栈

- The Stack ADT stores arbitrary objects
- Insertions and deletions follow the **last-in first-out scheme**

Think of a deck of cards. Main stack operations:

- **push(object):** inserts an element
- **object pop():** removes and returns the last inserted element

2. The Stack ADT

- Auxiliary stack operations:
- object **top()**: returns the last inserted element without removing it
- integer **len()**: returns the number of elements stored
- boolean **is_empty()**: indicates whether no elements are stored

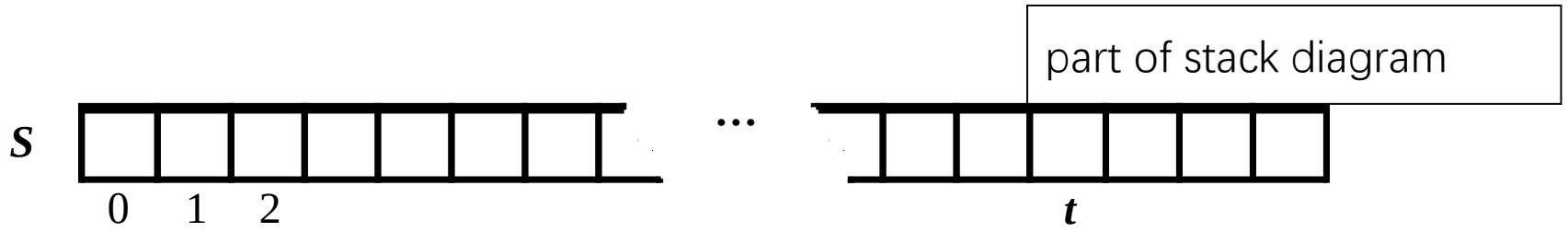
3. Stack

Array-based Stack

A simple way of implementing the Stack ADT uses an array

We add elements from left to right

A variable keeps track of the index of the top element



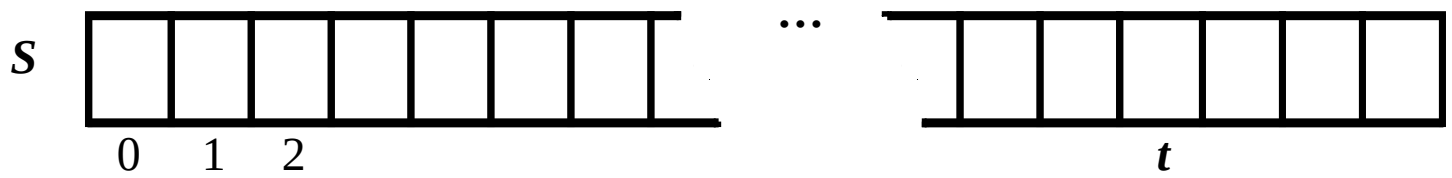
華南師範大學
SOUTH CHINA NORMAL UNIVERSITY
UNIVERSITY OF ABERDEEN

3. Stack

Array-based Stack (Cont.)

The array storing the stack elements may become full

A push operation will then need to grow the array and copy all the elements over



3. Stack

Example:

Operation	Return Value	Stack Contents
S.push(5)	–	[5]
S.push(3)	–	[5, 3]
len(S)	2	[5, 3]
S.pop( )	3	[5]
S.is_empty( )	False	[5]
S.pop( )	5	[ ]
S.is_empty( )	True	[ ]
S.pop( )	"error"	[ ]
S.push(7)	–	[7]
S.push(9)	–	[7, 9]
S.top( )	9	[7, 9]
S.push(4)	–	[7, 9, 4]
len(S)	3	[7, 9, 4]
S.pop( )	4	[7, 9]
S.push(6)	–	[7, 9, 6]
S.push(8)	–	[7, 9, 6, 8]
S.pop( )	8	[7, 9, 6]

```
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
```

UNIVERSITY OF ABERDEEN

```
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
```

4. The Queue ADT

- The Queue ADT stores arbitrary objects
- **Insertions and deletions follow the first-in first-out scheme**
- Insertions are at the rear of the queue and removals are at the front of the queue
- Main queue operations:
  - enqueue(object): inserts an element at the end of the queue
  - object dequeue(): removes and returns the element at the front of the queue

4. The Queue ADT

Operation	Return Value	first ← Q ← last
Q.enqueue(5)	–	[5]
Q.enqueue(3)	–	[5, 3]
len(Q)	2	[5, 3]
Q.dequeue( )	5	[3]
Q.is_empty( )	False	[3]
Q.dequeue( )	3	[ ]
Q.is_empty( )	True	[ ]
Q.dequeue( )	"error"	[ ]
Q.enqueue(7)	–	[7]
Q.enqueue(9)	–	[7, 9]
Q.first( )	7	[7, 9]
Q.enqueue(4)	–	[7, 9, 4]
len(Q)	3	[7, 9, 4]
Q.dequeue( )	7	[9, 4]

A rough implementation

```
class Queue:
    def __init__(self):
        self.items = []

    def isEmpty(self):
        return self.items == []

    def enqueue(self, item):
        self.items.insert(0,item)

    def dequeue(self):
        return self.items.pop()

    def size(self):
        return len(self.items)
```

http://interactivepython.org/runestone/static/pythonds/BasicDS/ImplementingaQueueinPython.html

This Not efficient!

Why? Insert(0, item) is very expensive.

```
N=100
k=0
data=[]
for n in range(N):
  data.insert(k, None)
```

	N				
	100	1,000	10,000	100,000	1,000,000
$k = 0$	0.482	0.765	4.014	36.643	351.590
$k = n // 2$	0.451	0.577	2.191	17.873	175.383
$k = n$	0.420	0.422	0.395	0.389	0.397

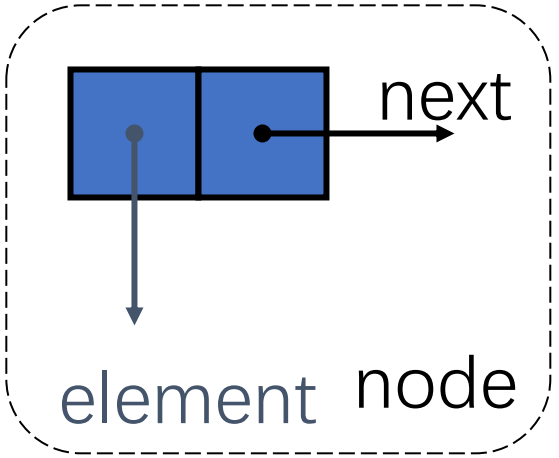
Goodrich book Sec 5.4.1

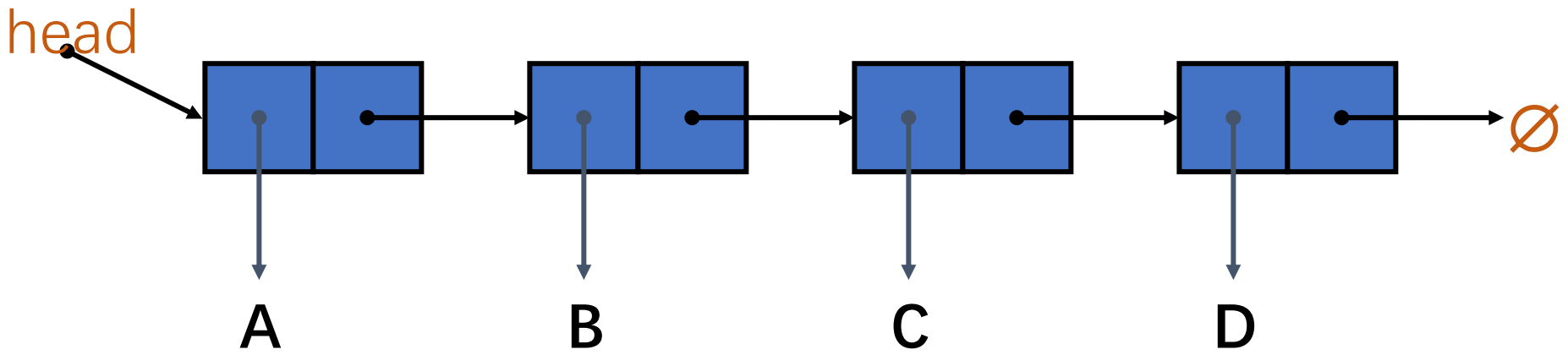
UNIVERSITY OF ABERDEEN

Linked List

1. 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**



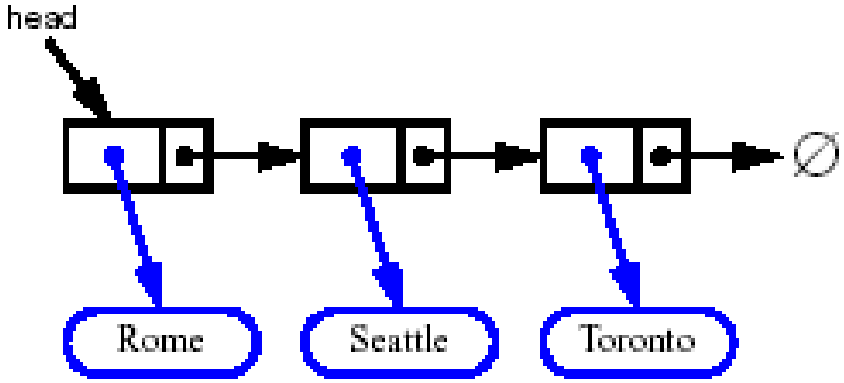


2. The Node Class for List Nodes

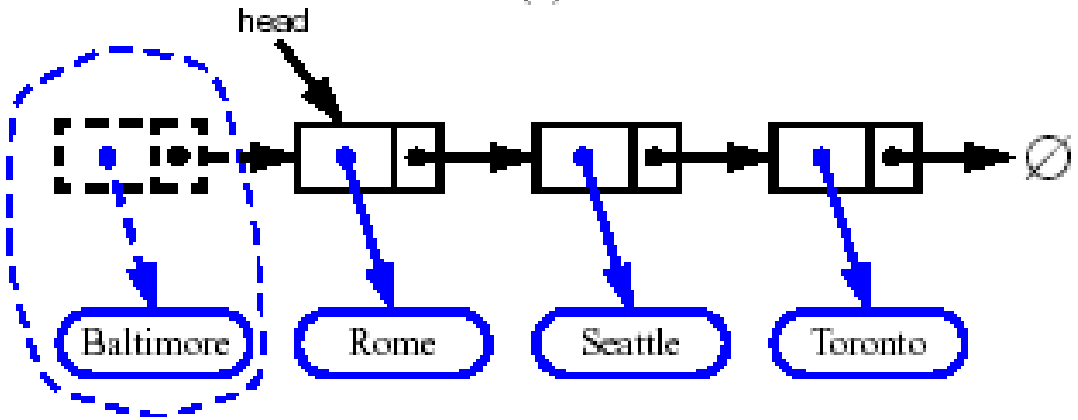
```
class _Node:
  """Lightweight, nonpublic class for storing a singly linked node."""
  __slots__ = '_element', '_next'          # streamline memory usage

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

3. Inserting at the Head

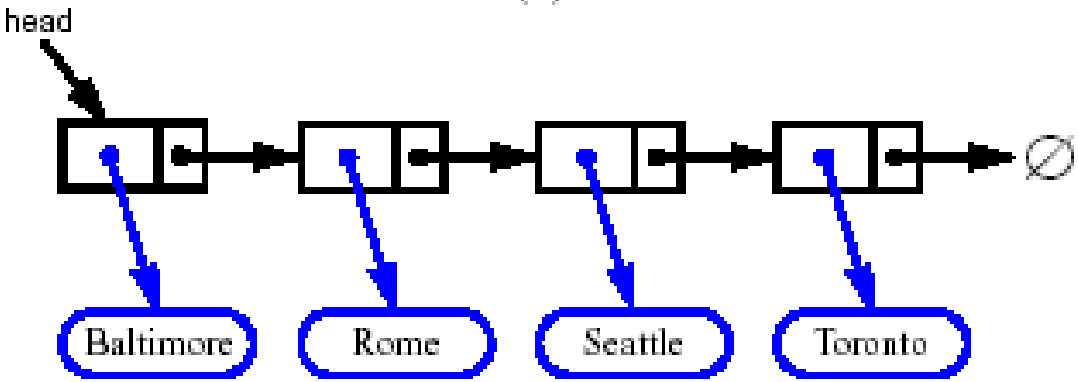
1. Allocate a new node
2. Insert new element
3. Have new node point to old head
4. Update head to point to new node



(a)



(b)



(c)

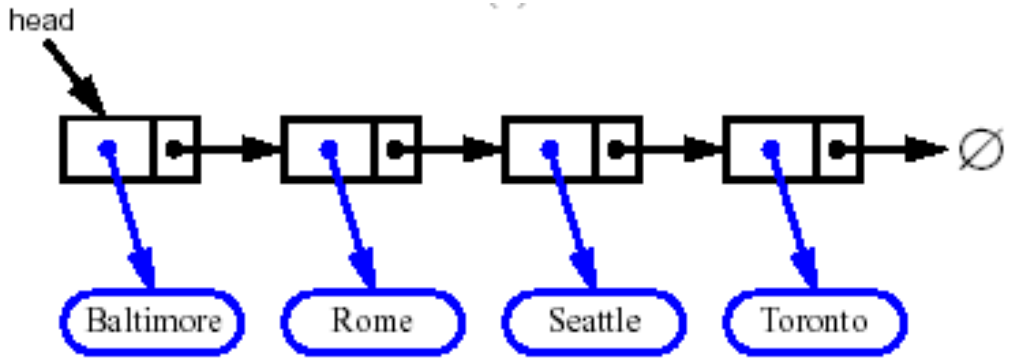
Algorithm add_first(L, e):

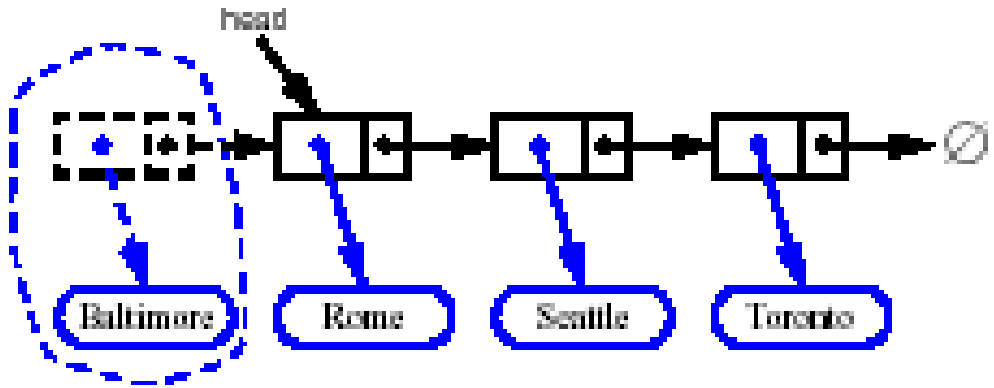
```
newest = Node(e)  {creat
newest.next = L.head    {
L.head = newest
L.size = L.size + 1
```

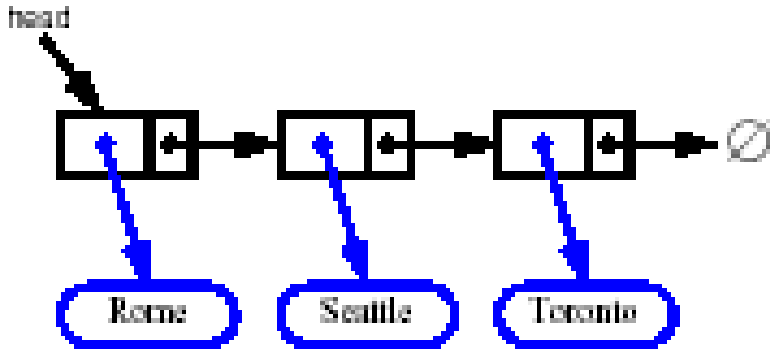
華南師範大學
SOUTH CHINA NORMAL UNIVERSITY

4. 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.

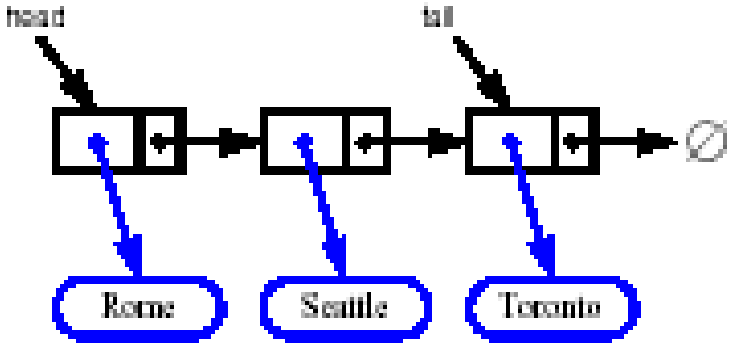
L.head = L.head.next {m

L.size = L.size − 1

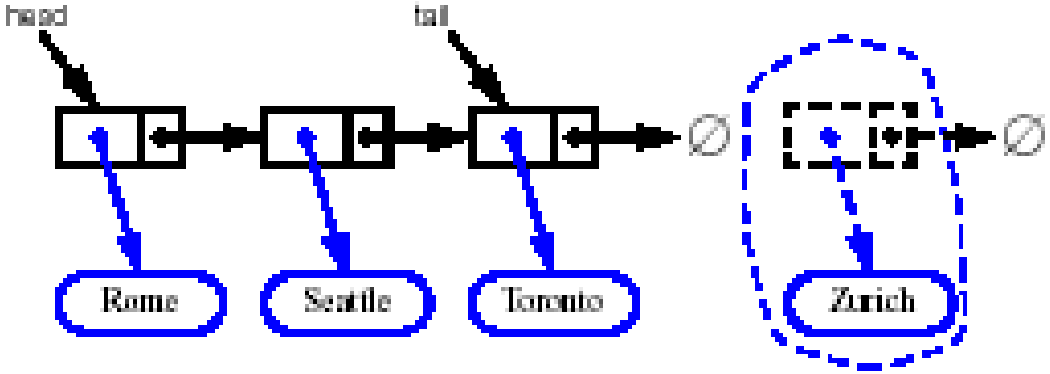
華南師範大學
SOUTH CHINA NORMAL UNIVERSITY
ABERDEEN

5. Inserting at the Tail

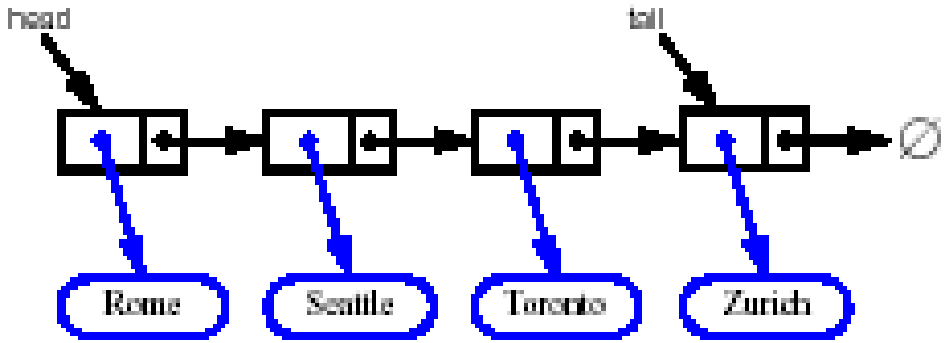
1. Allocate a new node
2. Insert new element
3. Have new node point to null
4. Have old last node point to new node
5. Update tail to point to new node



(a)



(b)



(c)

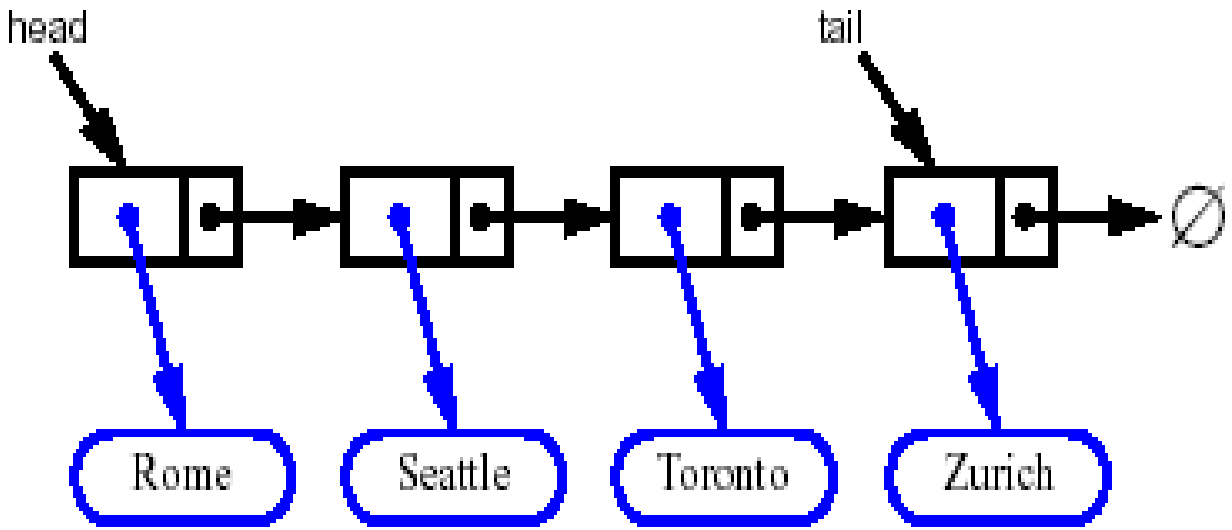
Algorithm add_last(L, e):

```
newest = Node(e)  {c
newest.next = None
L.tail.next = newest
L.tail = newest
L.size = L.size + 1
```

华南师范大学 SOUTH CHINA NORMAL
UNIVERSITY OF ABERDEEN

6. Removing at the Tail

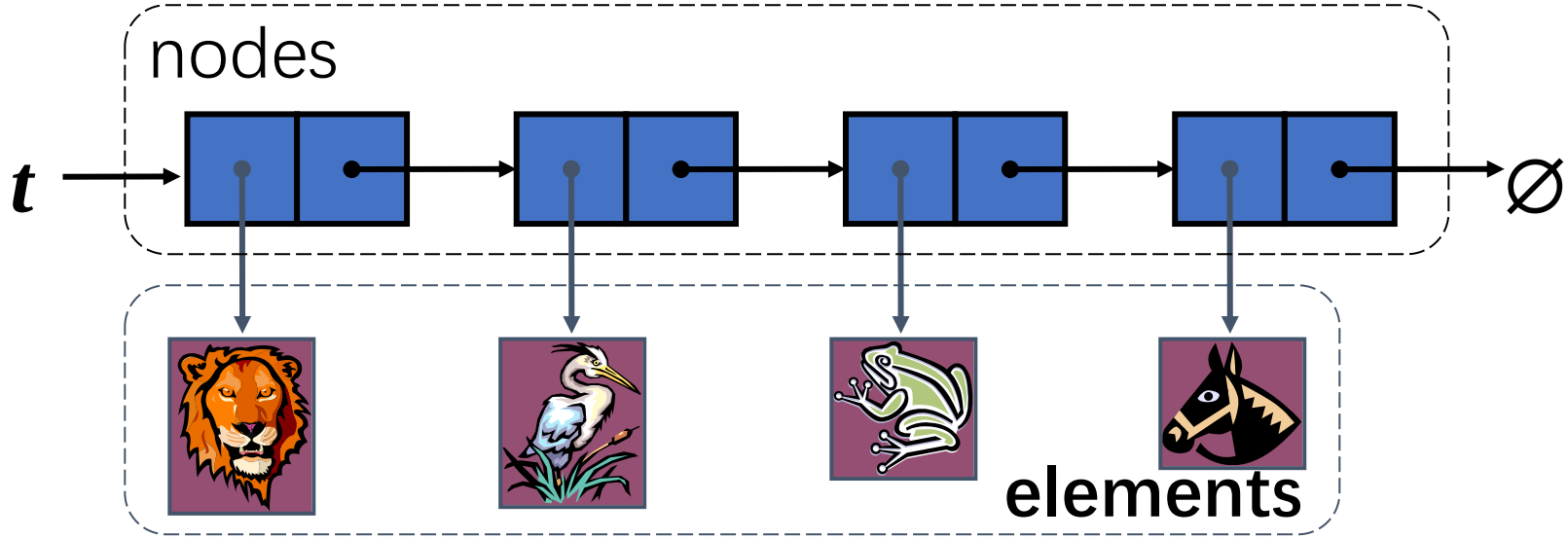
- Removing at the tail of a singly linked list is not efficient!
- There is no constant-time way to update the tail to point to the previous node



Stack as a Linked List

1. 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



2. Linked-

```
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
```

2. Linked-List Stack in Python

```
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) t
34
35      Raise Empty exception if the stac
36      """
37      if self.is_empty( ):
38        raise Empty('Stack is empt
39      return self._head._element
```

```
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
```

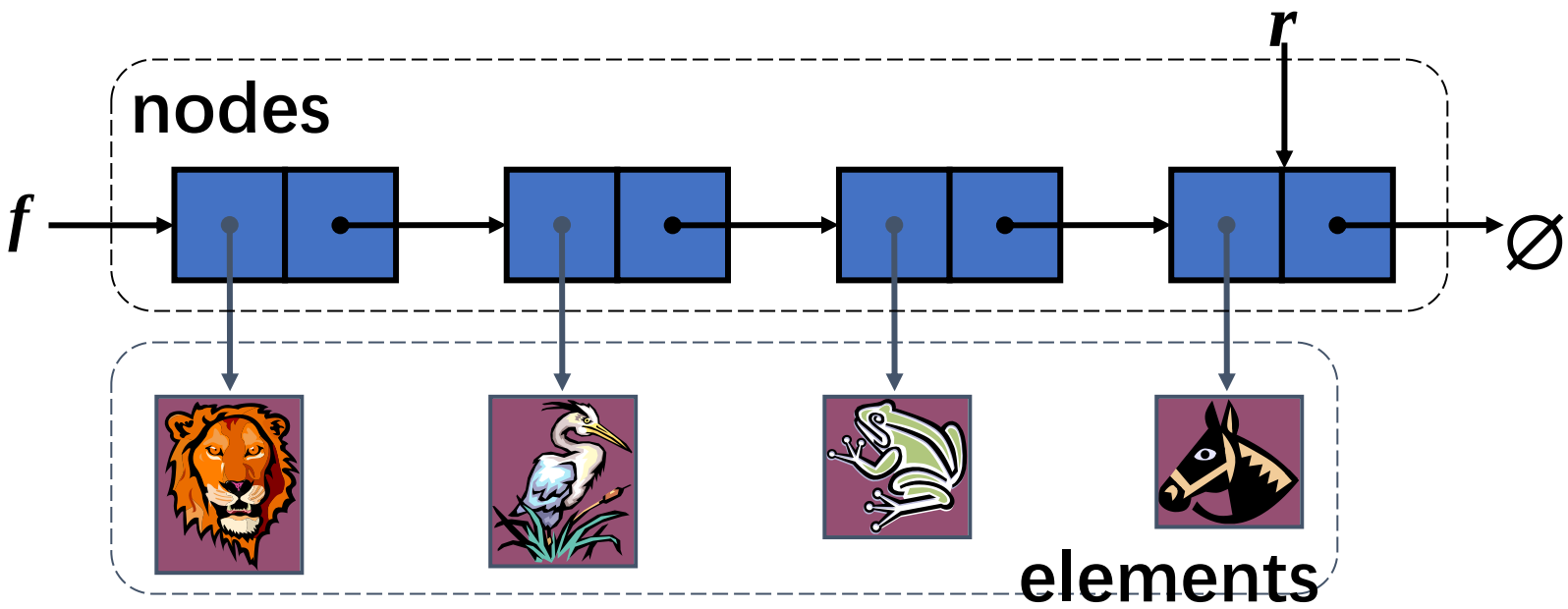
3. Linked-List Stack in Python

```
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
```

Queue as a Linked List

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



1. Queue

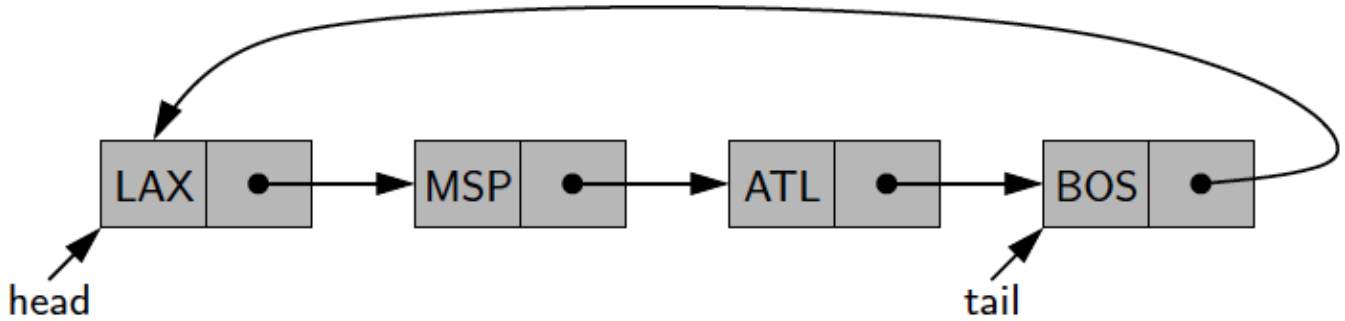
```
1  class LinkedQueue:
2    """FIFO queue implementation using a singly linked list for storage."""
3
4    class _Node:
5      """Lightweight, nonpublic class for storing a singly linked node."""
6      (omitted here; identical to that of LinkedStack._Node)
7
8    def __init__(self):
9      """Create 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
```

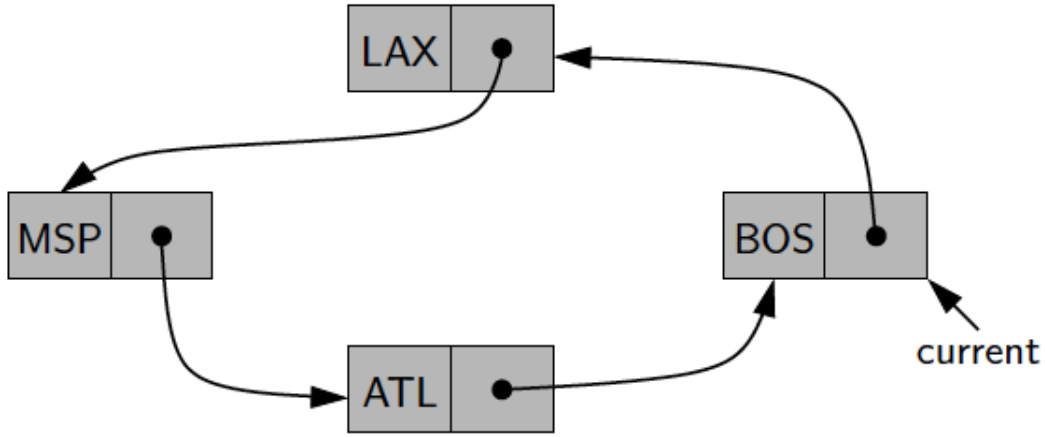
華南師範大學 SOUTH CHINA NORMAL UNIVERSITY
UNIVERSITY OF ABERDEEN

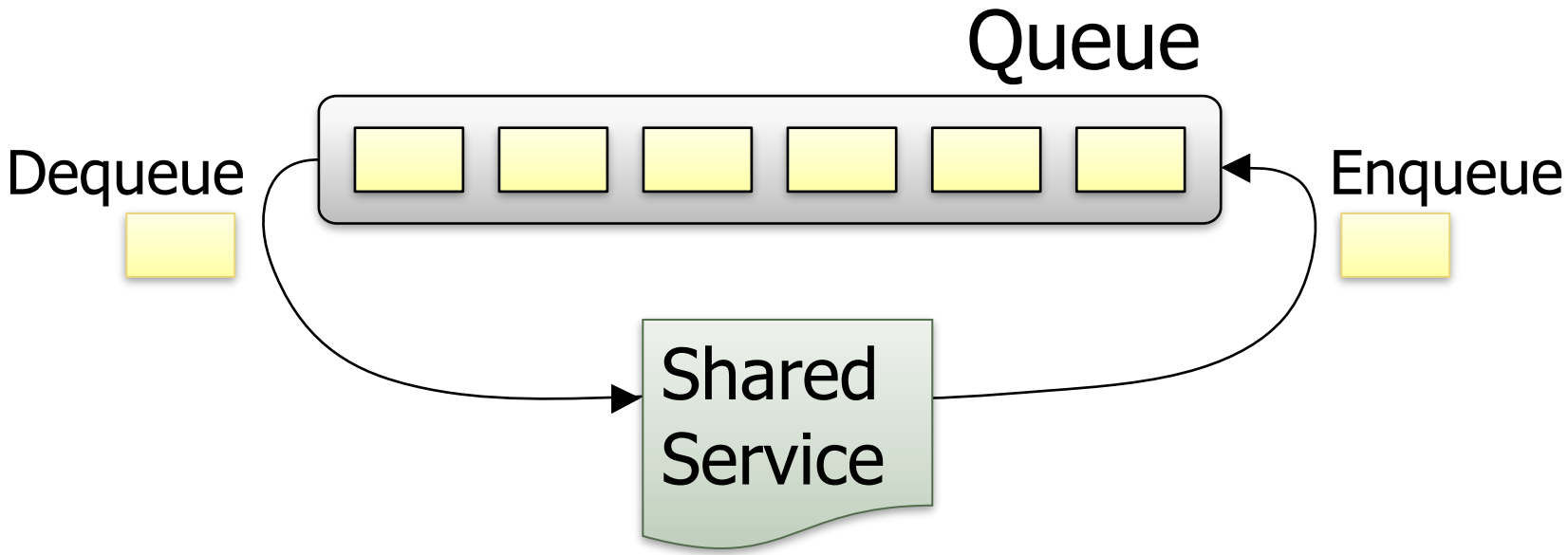
```
27      def dequeue(self):
28          """Remove and return the first element of the queue (i.e., FIFO).
29
30          Raise Empty exception if the queue is empty.
31          """
32          if self.is_empty( ):
33              raise Empty('Queue is empty')
34          answer = self._head._element
35          self._head = self._head._next
36          self._size -= 1
37          if self.is_empty( ):                    # special case as queue is em
38              self._tail = None                   # removed head had been th
39          return answer
```

```
41      def enqueue(self, e):
42        """Add an element to the back of queue."""
43        newest = self._Node(e, None)           # node will
44        if self.is_empty( ):
45          self._head = newest                  # special cas
46        else:
47          self._tail._next = newest
48        self._tail = newest                    # update ref
49        self._size += 1
```

Circularly Linked List

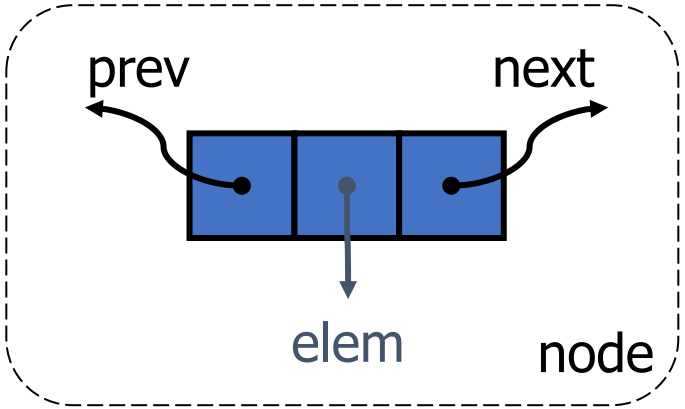


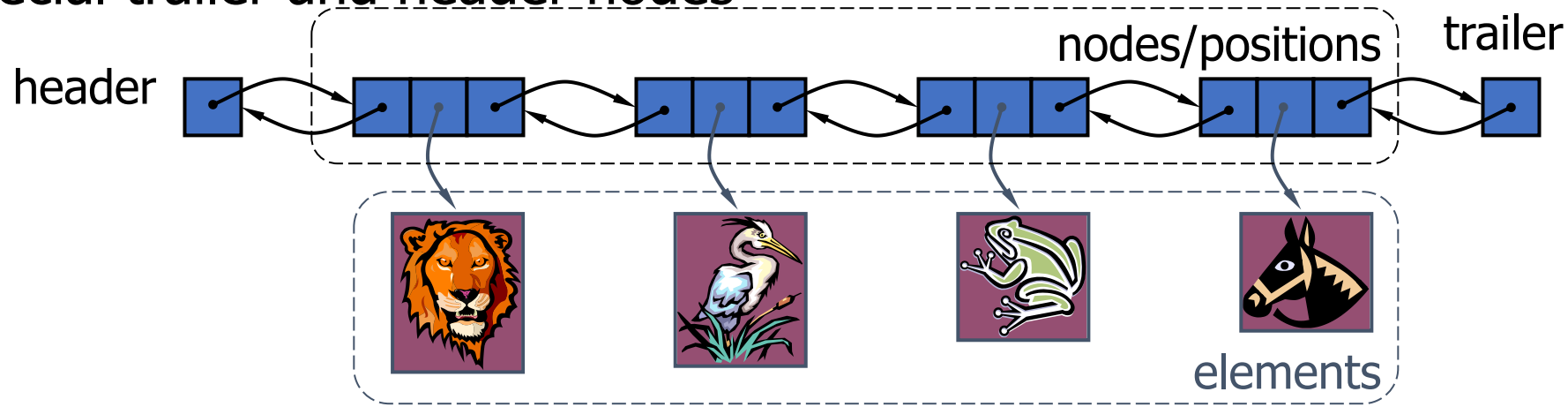




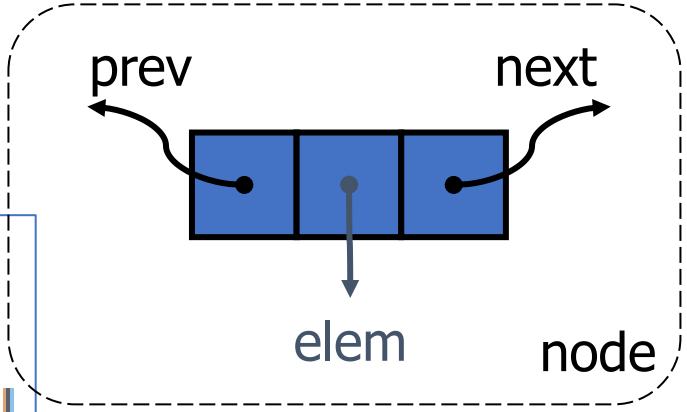
Doubly Linked List

- 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



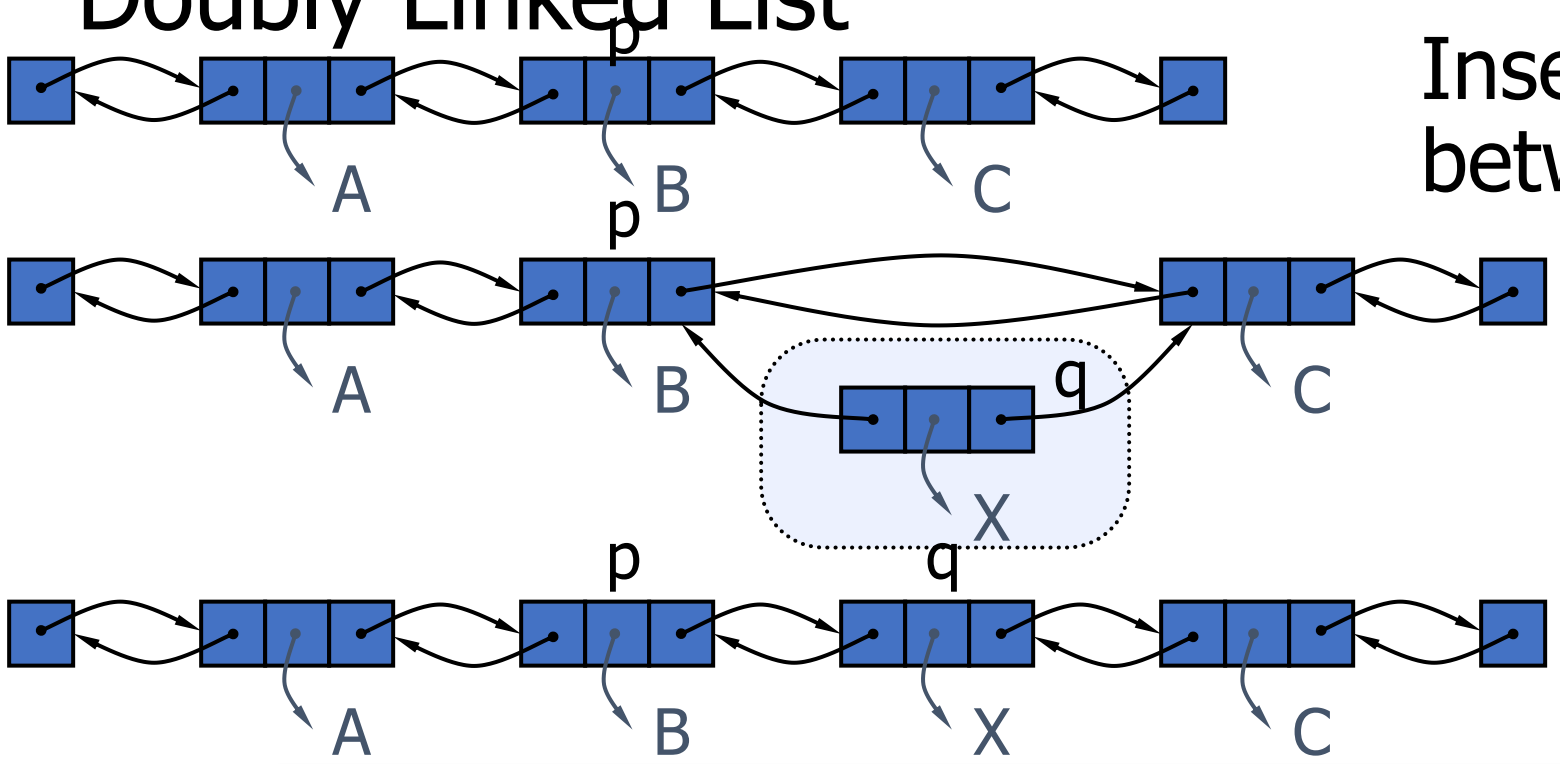


Doubly Linked List




```
class _Node:
  """Lightweight, nonpublic class for storing a do
  __slots__ = '_element', '_prev', '_next'

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

Doubly Linked List



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

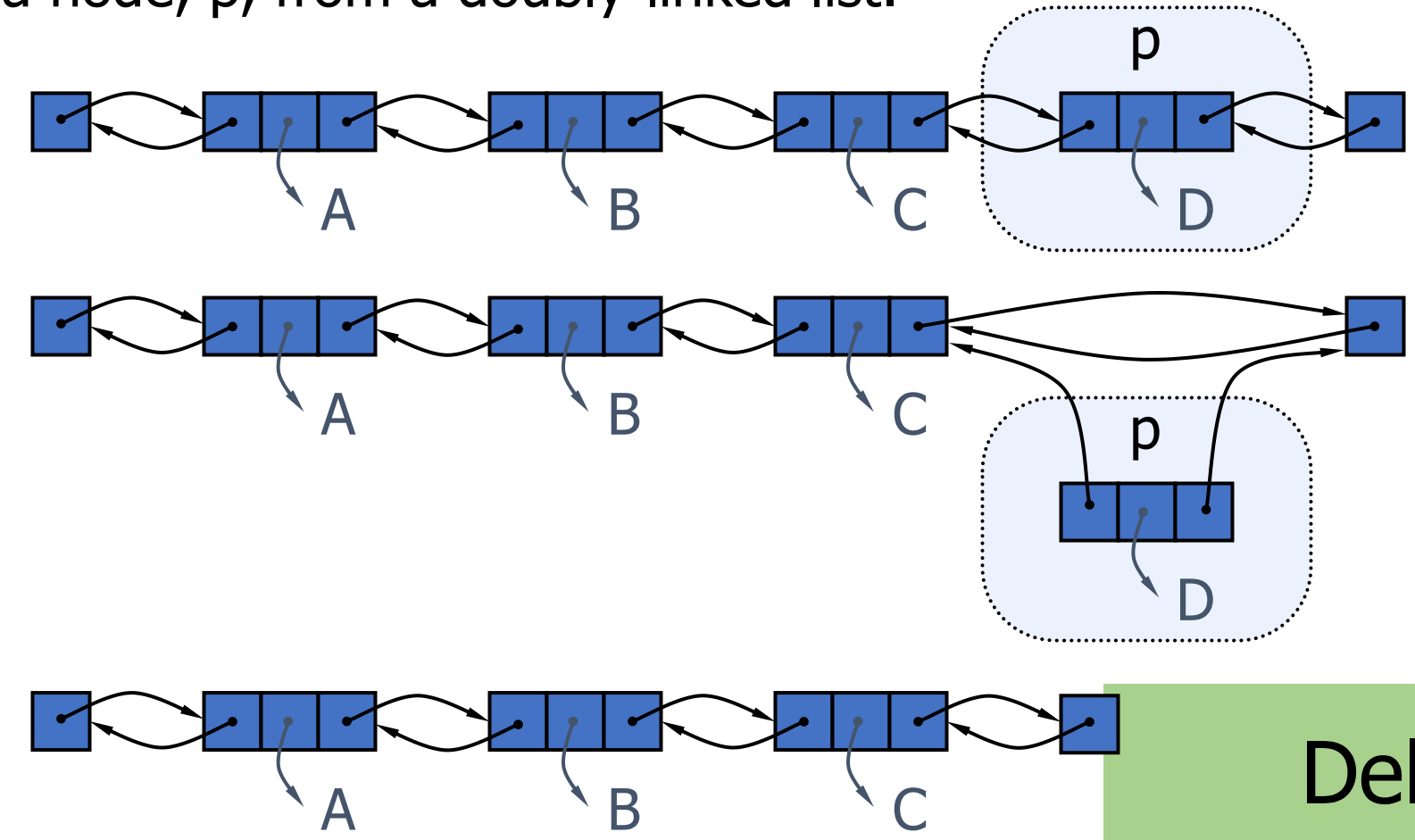
Insertion

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

SOUTH CHINA NORM

Doubly Linked List

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



Deletion

華南師範大學
SOUTH CHINA NORMAL UNIVERSITY
UNIVERSITY OF ABERDEEN

Doubly Linked List

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

```
32    def _delete_node(self, node):
33      """Delete nonsentinel node from the list and return its element."""
34      predecessor = node._prev
35      successor = node._next
36      predecessor._next = successor
37      successor._prev = predecessor
38      self._size -= 1
39      element = node._element                    # record deleted element
40      node._prev = node._next = node._element = None     # deprecate node
41      return element                             # return deleted element
```

Deletion

Thank You