武汉大学
WUHAN UNIVERSITY

数据结构与算法

链表

张晓平

武汉大学数学与统计学院

Table of contents

单链表 (Singly Linked Lists)

单链表 (Singly Linked Lists)

定义：单链表

单链表是一组节点的集合，这些节点共同构成一个线性序列。每个节点存储对作为序列元素对象的引用，以及对列表的下一个节点的引用。

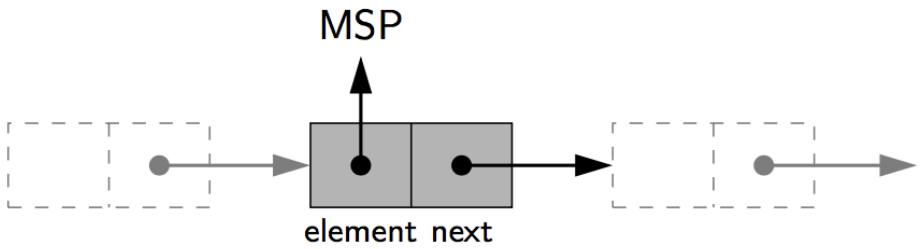


Figure 7.1: Example of a node instance that forms part of a singly linked list. The node's element member references an arbitrary object that is an element of the sequence (the airport code MSP, in this example), while the next member references the subsequent node of the linked list (or None if there is no further node).

单链表 (Singly Linked Lists)

定义：表头与表尾 (head and tail)

链表的第一个和最后一个节点分别称为表头和表尾。

定义：遍历 (traverse)

从表头开始，通过跟踪每个节点的下一个引用从一个节点移动到另一个节点，便可到达表尾。表尾节点为直接后继为 None 的节点。这个过程通常称为遍历链表。

单链表 (Singly Linked Lists)

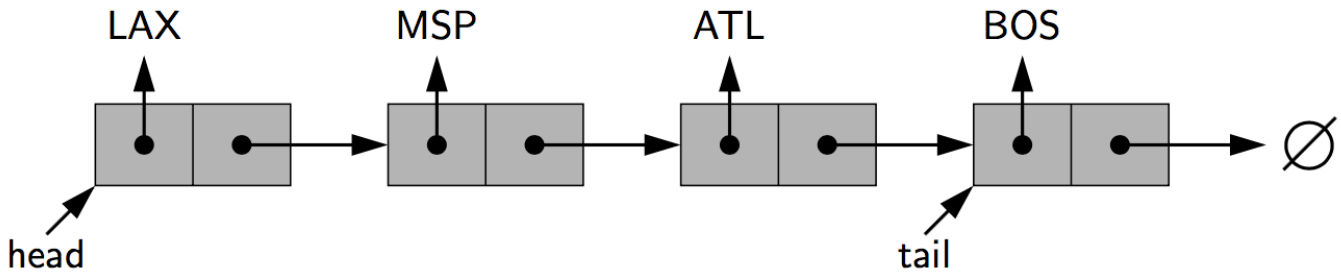


Figure 7.2: Example of a singly linked list whose elements are strings indicating airport codes. The list instance maintains a member named head that identifies the first node of the list, and in some applications another member named tail that identifies the last node of the list. The **None** object is denoted as Ø.

单链表 (Singly Linked Lists)

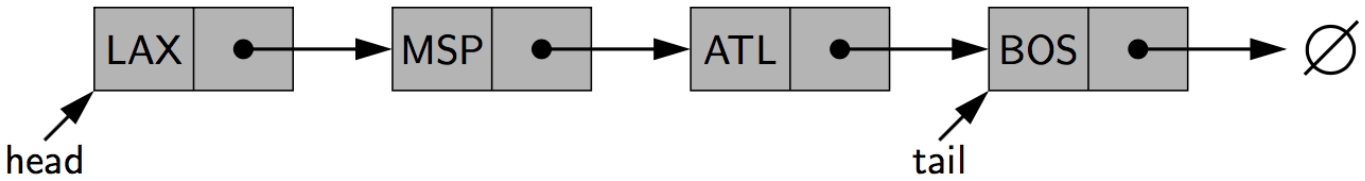


Figure 7.3: A compact illustration of a singly linked list, with elements embedded in the nodes (rather than more accurately drawn as references to external objects).

单链表 (Singly Linked Lists)

向表头插入元素

向表头插入元素

链表的一个重要特性是它没有预定的固定大小；它使用与其当前元素数量成比例的空间。

向表头插入元素

问题

如何在表头处插入元素呢?

向表头插入元素

问题

如何在表头处插入元素呢?

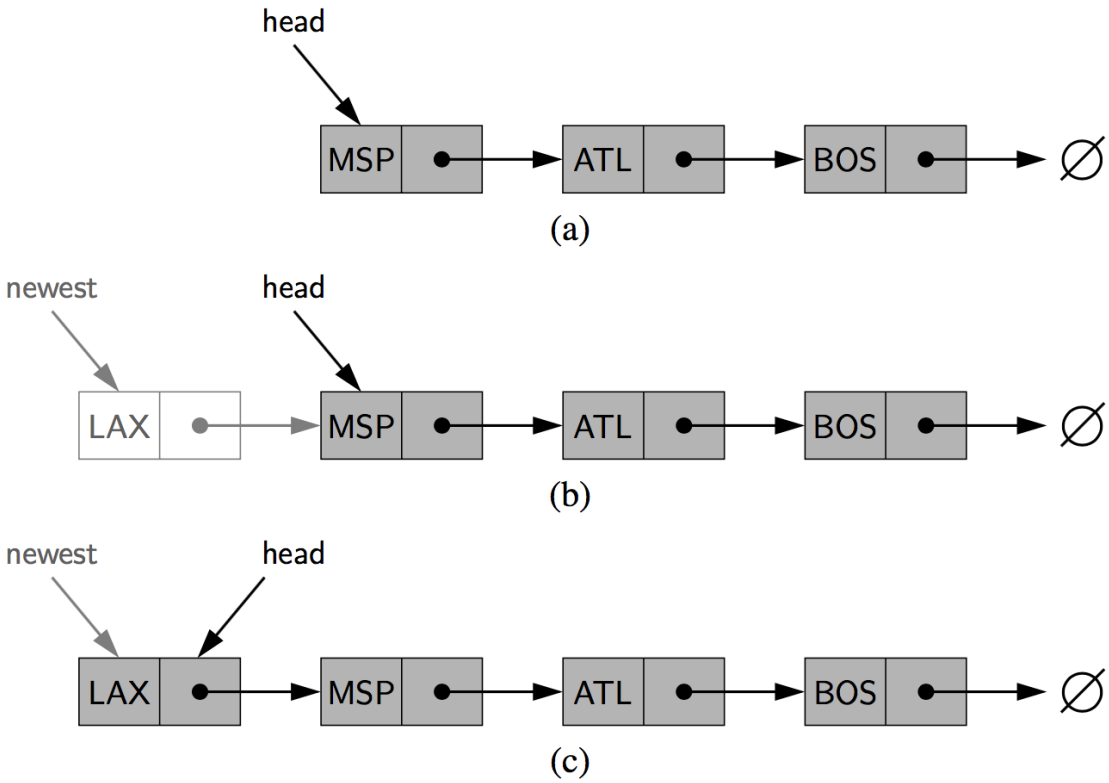


Figure 7.4: Insertion of an element at the head of a singly linked list: (a) before the insertion; (b) after creation of a new node; (c) after reassignment of the head reference.

向表头插入元素

1. 创建新节点

   将其元素设置为新元素；

   设置当前表头为其直接后继；

2. 将表头设置为指向新节点。

向表头插入元素

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.

单链表 (Singly Linked Lists)

向表尾插入元素

向表尾插入元素

还可以很容易地在表尾插入一个元素，只要我们保留对尾部节点的引用。

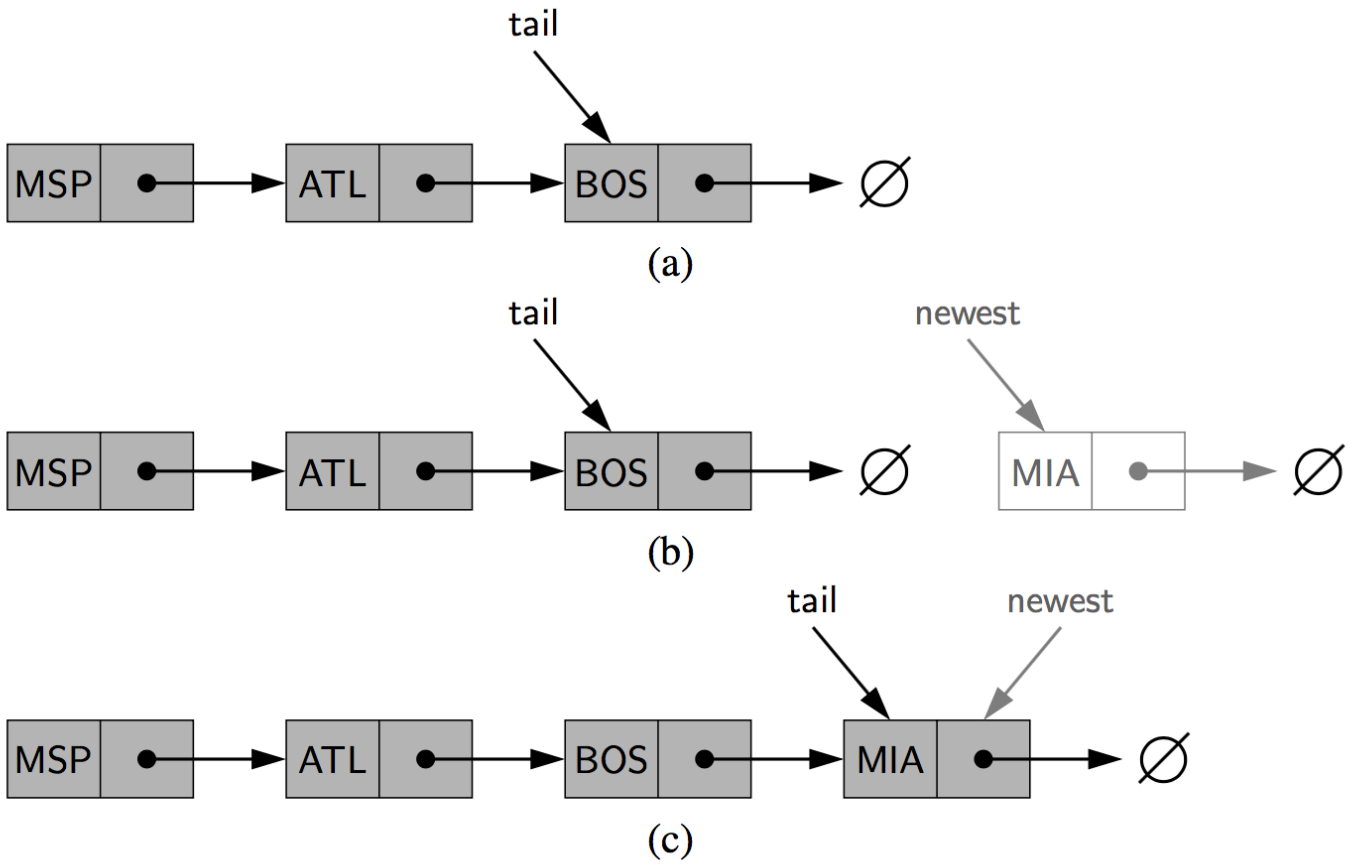


Figure 7.5: Insertion at the tail of a singly linked list: (a) before the insertion; (b) after creation of a new node; (c) after reassignment of the tail reference. Note that we must set the next link of the tail in (b) before we assign the tail variable to point to the new node in (c).

向表尾插入元素

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.

单链表 (Singly Linked Lists)

删除表头元素

删除表头元素

从表头删除一个元素实质上是在表头处插入元素的反向操作。

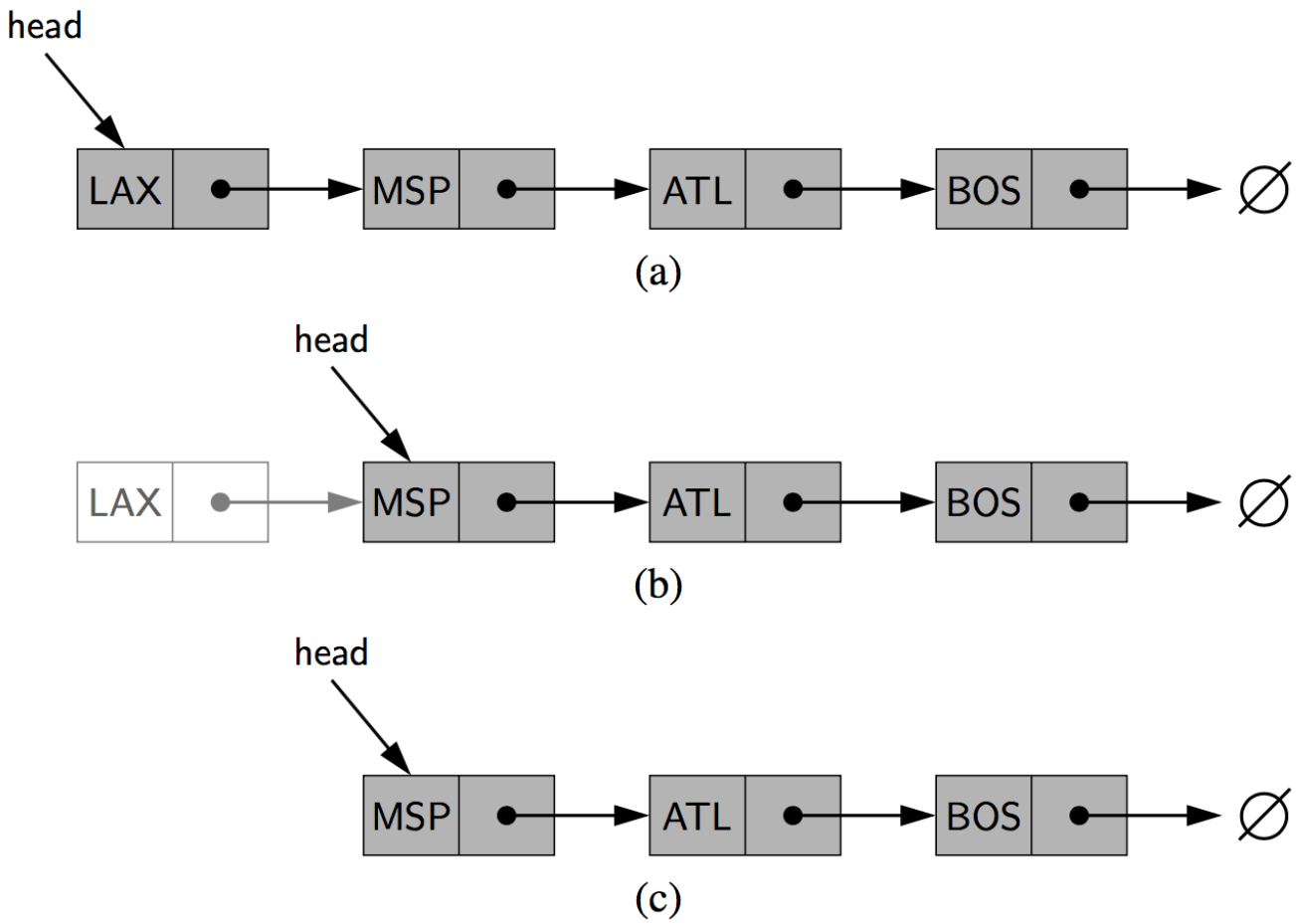


Figure 7.6: Removal of an element at the head of a singly linked list: (a) before the removal; (b) after "linking out" the old head; (c) final configuration.

单链表 (Singly Linked Lists)

删除表尾元素

删除表尾元素

不幸的是，删除表尾节点并不容易。

- 即使我们直接维护到列表最后一个节点的尾部引用，我们也必须能够访问最后一个节点之前的节点，以便删除最后一个节点。
- 但是我们不能通过跟踪尾部的下一个链接来到达尾部之前的节点。
- 访问此节点的唯一方法是从列表的头部开始并一直搜索列表但这样一系列的跳频操作可能需要很长时间。

删除表尾元素

不幸的是，删除表尾节点并不容易。

- 即使我们直接维护到列表最后一个节点的尾部引用，我们也必须能够访问最后一个节点之前的节点，以便删除最后一个节点。
- 但是我们不能通过跟踪尾部的下一个链接来到达尾部之前的节点。
- 访问此节点的唯一方法是从列表的头部开始并一直搜索列表但这样一系列的跳频操作可能需要很长时间。

如果我们想有效地支持这样一个行动，需要用到双链表 (doubly linked list).

单链表 (Singly Linked Lists)

使用单链表实现堆栈

使用单链表实现堆栈

使用单链表实现堆栈，需确定是用表头还是表尾作为栈顶。

使用单链表实现堆栈

使用单链表实现堆栈，需确定是用表头还是表尾作为栈顶。

可将栈顶定位表头，因为这样可以有效地在表头处以恒定时间插入和删除元素，而所有堆栈操作都会影响顶部。

使用单链表实现堆栈

```
from exceptions import Empty
class LinkedStack:

  class _Node:
    __slots__ = '_element', '_next'
    def __init__(self, element, next):
      self._element = element
      self._next = next

  def __init__(self):
    self._head = None
    self._size = 0

  def __len__(self):
```

使用单链表实现堆栈

```
    return self._size

  def is_empty(self):
    return self._size == 0

  def push(self, e):
    node = self._Node(e, self._head)
    self._head = node
    self._size += 1

  def pop(self):
    if self.is_empty():
      raise Empty('Stack is empty')
    self.head
```

使用单链表实现堆栈

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

Table 7.1: Performance of our LinkedStack implementation. All bounds are worst-case and our space usage is $O(n)$, where n is the current number of elements in the stack.

单链表 (Singly Linked Lists)

使用单链表实现队列

使用单链表实现队列

因为队列操作在两端执行，所以我们将显式地维护表头引用和表尾引用作为每个队列的实例变量。

队列的自然方向是将队头与表头部对齐，将队尾与列尾对齐。这样可以保证新进的元素在队尾入队列，队列元素从队头出队列。

使用单链表实现队列

```
from exceptions import Empty
class LinkedQueue:
  class _Node:
    __slots__ = '_element', '_next'
    def __init__(self, element, next):
      self._element = element
      self._next = next
  def __init__(self):
    self._head = None
    self._tail = None
    self._size = 0
  def __len__(self):
    return self._size
  def is_empty(self):
    return self._size == 0
```

使用单链表实现队列

```
def first(self):
  if self.is_empty():
    raise Empty('Queue is empty')
  return self._head._element
def dequeue(self):
  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():
    self._tail = None
  return answer
```

使用单链表实现队列

```
def enqueue(self, e):
    newest = self._Node(e, None)
    if self.is_empty():
        self._head = newest
    else:
        self._tail._next = newest
    self._tail = newest
    self._size += 1
```

使用单链表实现队列

在性能方面，LinkedQueue 与 LinkedStack 类似，所有操作都在最坏情况下的常量时间内运行，并且空间使用量在当前元素数中是线性的。

循环链表 (Circularly Linked Lists)

循环链表 (Circularly Linked Lists)

定义：循环链表

循环链表是一个链表，其表尾节点的直接后继为其表头节点。

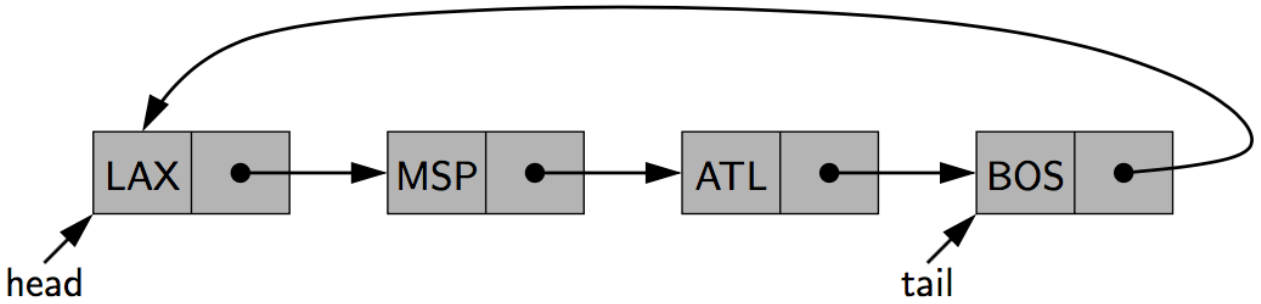


Figure 7.7: Example of a singly linked list with circular structure.

循环链表 (Circularly Linked Lists)

对于循环类型 (即没有开始和结束) 的数据集，循环链表比标准链表更通用。

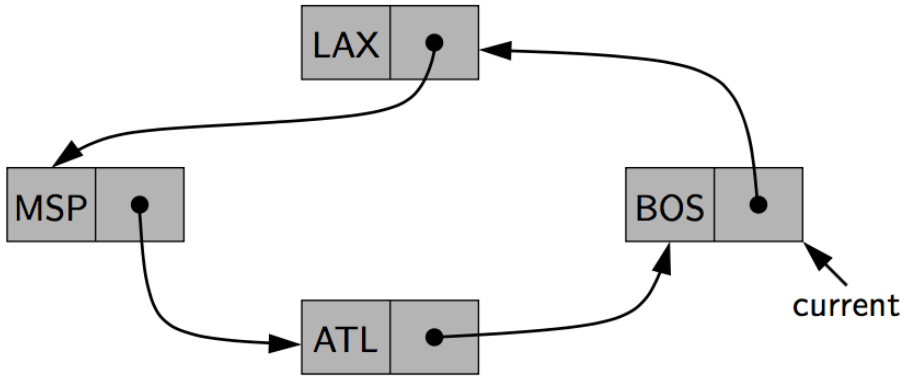


Figure 7.8: Example of a circular linked list, with current denoting a reference to a select node.

尽管循环链表本身没有开始或结束，但必须维护一个指定节点的引用，通常用标识符 current 表示。这样就可以使用 current = current.next 移动到下一个节点。

循环链表 (Circularly Linked Lists)

使用循环链表实现队列

使用循环链表实现队列

```
from exceptions import Empty
class CircularQueue:
  class _Node:
    __slots__ = '_element', '_next'
    def __init__(self, element, next):
      self._element = element
      self._next = next
  def __init__(self):
    self._tail = None
    self._size = 0
  def __len__(self):
    return self._size
  def is_empty(self):
    return self._size == 0
```

使用循环链表实现队列

```
def first(self):
  if self.is_empty():
    raise Empty('Queue is empty')
  head = self._tail._next
  return head._element
def dequeue(self):
  if self.is_empty():
    raise Empty('Queue is empty')
  oldhead = self._tail._next
  if self._size == 1:
    self._tail = None
  else:
    self._tail._next = oldhead._next
  self._size -= 1
  return oldhead._element
```

使用循环链表实现队列

```
  def enqueue(self, e):
    newest = self._Node(e, None)
    if self.is_empty():
      newest._next = newest
    else:
      newest._next = self._tail._next
      self._tail._next = newest
    self._tail = newest
    self._size += 1
  def rotate(self):
    if self._size > 0:
      self._tail = self._tail._next
```

双重链表 (Doubly Linked Lists)

双重链表 (Doubly Linked Lists)

定义：双重链表

双重链表是一个链表，每个节点除了包含数据外，还包含对其直接前驱和直接后继的引用。

双重链表 (Doubly Linked Lists)

定义：双重链表

双重链表是一个链表，每个节点除了包含数据外，还包含对其直接前驱和直接后继的引用。

双重链表允许更丰富的常数时间操作，包括在链表的任意位置进行插入和删除。

双重链表 (Doubly Linked Lists)

定义：双重链表

双重链表是一个链表，每个节点除了包含数据外，还包含对其直接前驱和直接后继的引用。

双重链表允许更丰富的常数时间操作，包括在链表的任意位置进行插入和删除。

使用 next 来引用直接后继节点，并引入 prev 来引用直接前驱节点。

双重链表 (Doubly Linked Lists)

头部和表部岗哨 (Header and Trailer Sentinels)

头部和表部岗哨 (Header and Trailer Sentinels)

对于双重链表，处理边界节点时往往需要做一些特殊处理。若在链表的两端加入一些特殊节点会带来很多好处，即在链表的开始处加入头部节点，在结尾处加入尾部节点。这些“伪”节点也称为“岗哨”节点，它们并不存储数据。

头部和表部岗哨 (Header and Trailer Sentinels)

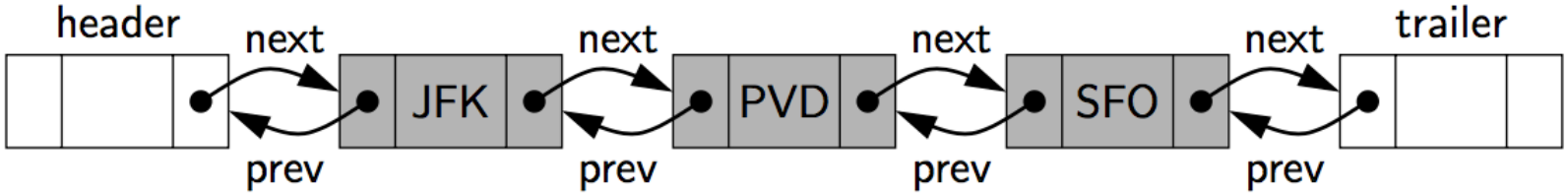


Figure 7.10: A doubly linked list representing the sequence { JFK, PVD, SFO }, using sentinels header and trailer to demarcate the ends of the list.

头部和表部岗哨 (Header and Trailer Sentinels)

使用岗哨节点时，

- 对于空链表，其头部节点的 prev 域指向 None，next 域指向尾部节点，其尾部节点的 prev 域指向头部节点，next 域指向 None。
- 对于非空链表，头部节点的 next 域指向表头节点，尾部节点的 prev 将指向表尾节点。

双重链表 (Doubly Linked Lists)

使用岗哨节点的好处

使用岗哨节点的好处

- 尽管不设置岗哨节点也可以实现双重链表，但是岗哨节点的引入会大大简化链表操作。

  请注意：俩岗哨节点永远不会改变，只有它们之间的节点才会改变。

- 有了岗哨节点，便可以用统一的方式对待所有插入和删除操作，因为插入节点总是被放置在两个已有节点之间，而欲删除节点的两侧都有邻居。

双重链表 (Doubly Linked Lists)

双重链表的插入和删除

双重链表的插入和删除

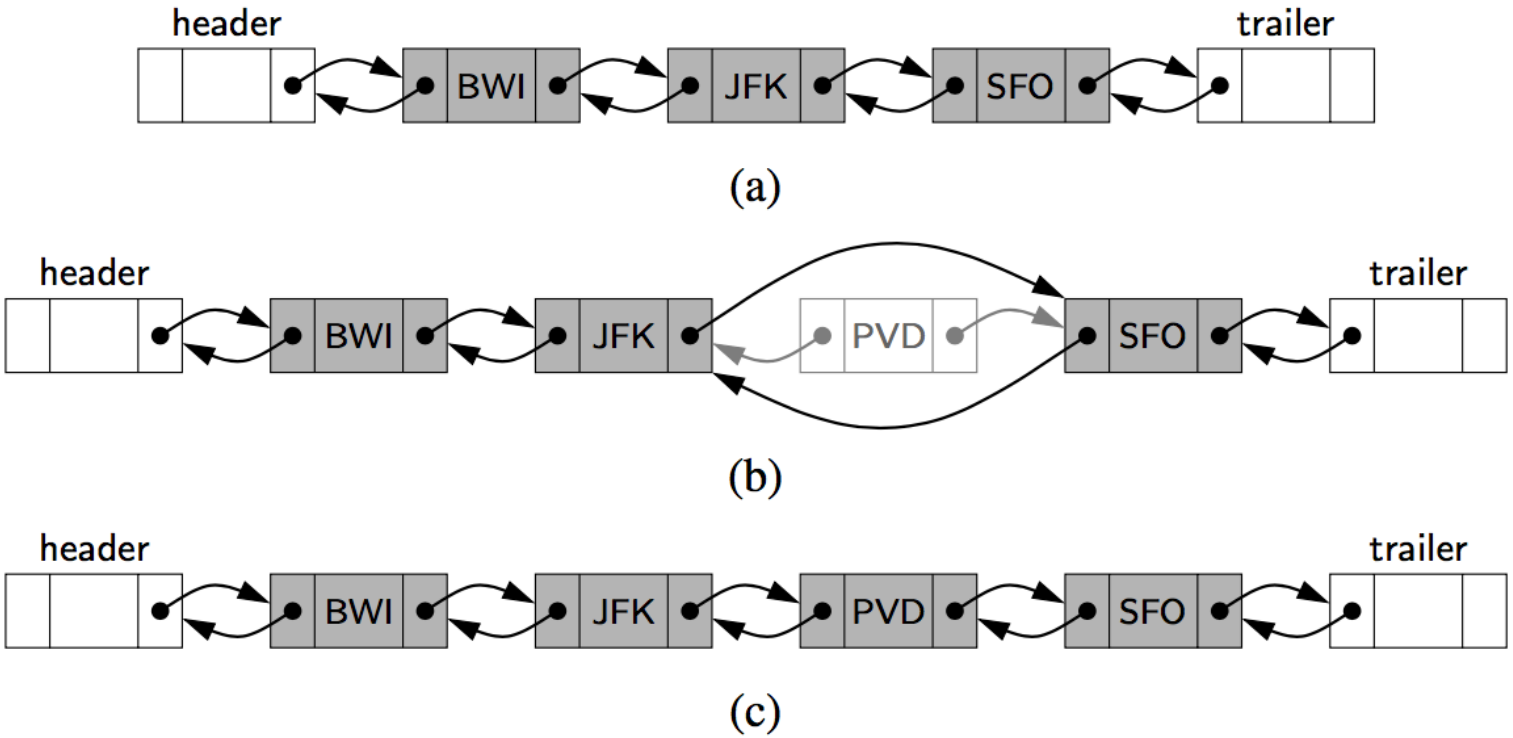


Figure 7.11: Adding an element to a doubly linked list with header and trailer sentinels: (a) before the operation; (b) after creating the new node; (c) after linking the neighbors to the new node.

双重链表的插入和删除

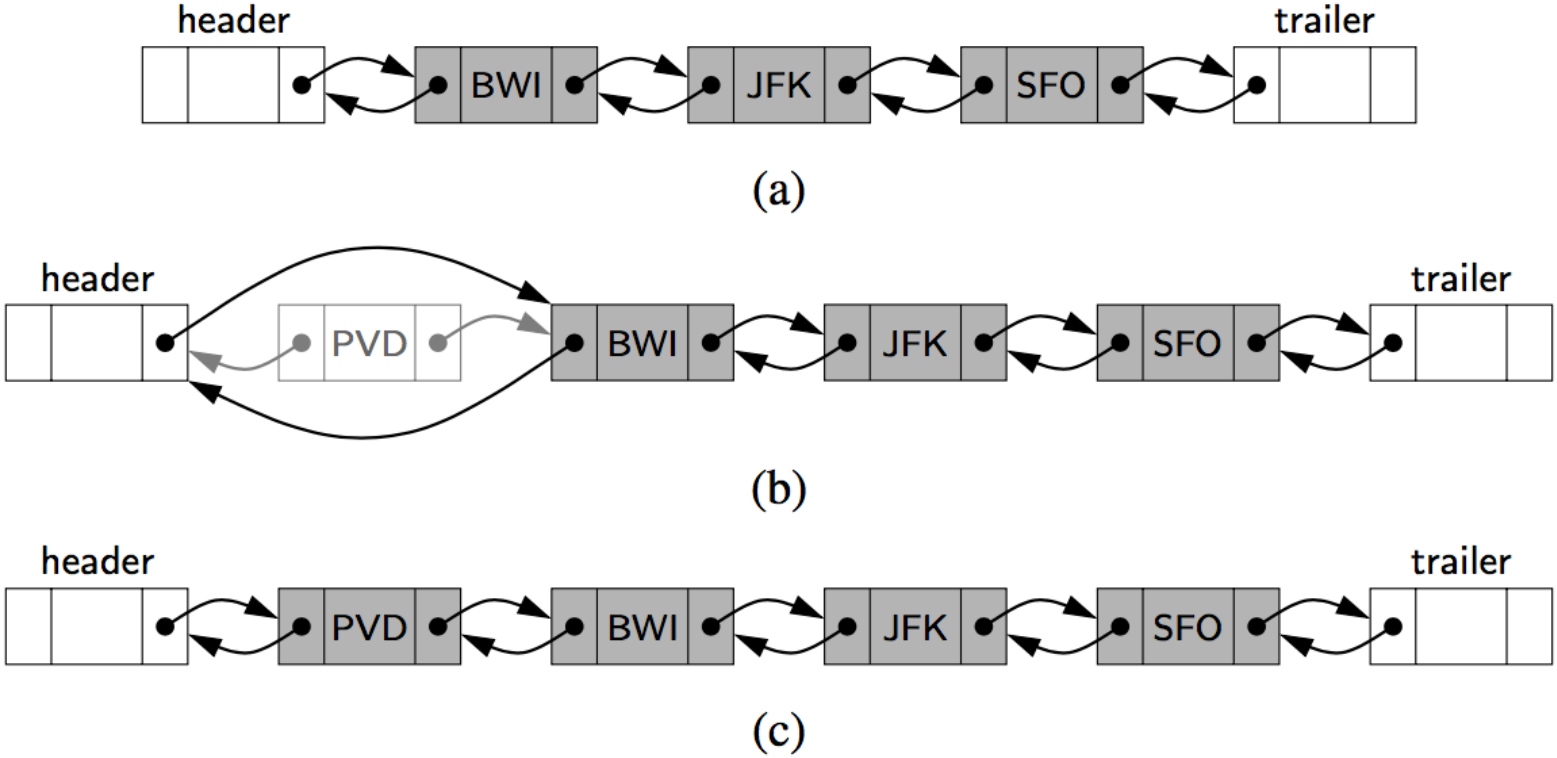


Figure 7.12: Adding an element to the front of a sequence represented by a doubly linked list with header and trailer sentinels: (a) before the operation; (b) after creating the new node; (c) after linking the neighbors to the new node.

双重链表的插入和删除

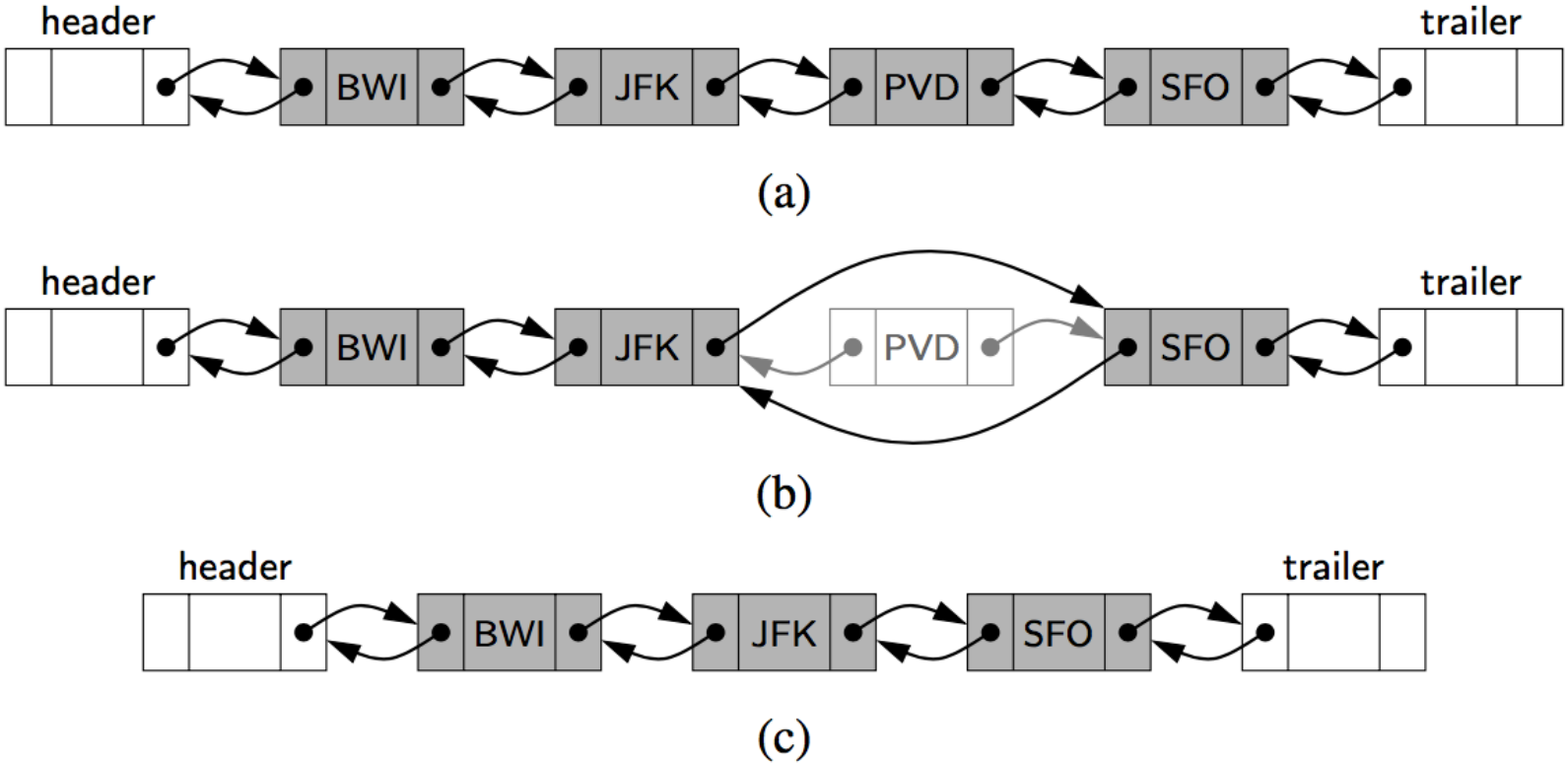


Figure 7.13: Removing the element PVD from a doubly linked list: (a) before the removal; (b) after linking out the old node; (c) after the removal (and garbage collection).

双重链表 (Doubly Linked Lists)

双重链表的基本实现

双重链表的基本实现

```
class _DoublyLinkedBase:
  class _Node:
    __slots__ = '_element', '_prev', '_next'
    def __init__(self, element, prev, next):
      self._element = element
      self._prev = prev
      self._next = next
  def __init__(self):
    self._header = self._Node(None, None, None)
    self._trailer = self._Node(None, None, None)
    self._header._next = self._trailer
    self._trailer._prev = self._header
    self._size = 0
  def __len__(self):
    return self._size
  def is_empty(self):
    return self._size == 0
```

双重链表的基本实现

```
def _insert_between(self, e, predecessor,
successor):
  newest = self._Node(e, predecessor,
  successor)
  predecessor._next = newest
  successor._prev = newest
  self._size += 1
  return newest
```

双重链表的基本实现

```
def _delete_node(self, node):
    predecessor = node._prev
    successor = node._next
    predecessor._next = successor
    successor._prev = predecessor
    self._size -= 1
    element = node._element
    node._prev = node._next = node._element =
    None
    return element
```

双重链表 (Doubly Linked Lists)

使用双重链表实现双端队列

对于双端队列 (deque)，

- 利用基于数组的实现，可在摊销意义下的常数时间内完成所有操作，因为有时需要调整数组的大小；
- 利用基于双链表的实现，可在最坏情况下的常数时间内实现所有操作。

使用双重链表实现双端队列

```
class LinkedDeque(_DoublyLinkedBase):
  def first(self):
    if self.is_empty():
      raise Empty("Deque is empty")
    return self._header._next._element
  def last(self):
    if self.is_empty():
      raise Empty("Deque is empty")
    return self._trailer._prev._element
```

使用双重链表实现双端队列

```
  def insert_first(self, e):
    self._insert_between(e, self._header, self.
    _header._next)
  def insert_last(self, e):
    self._insert_between(e, self._trailer._prev,
     self._trailer)
```

使用双重链表实现双端队列

```
  def delete_first(self):
    if self.is_empty():
      raise Empty("Deque is empty")
    return self._delete_node(self._header._next)
  def delete_last(self):
    if self.is_empty():
      raise Empty("Deque is empty")
    return self._delete_node(self._trailer._prev
    )
```

位置列表 ADT(The Positional List ADT)

位置列表 ADT(The Positional List ADT)

到目前为止，我们考虑过的抽象数据类型，即堆栈、队列和双端队列，只允许在序列的一端或另一端进行更新操作。

位置列表 ADT(The Positional List ADT)

到目前为止，我们考虑过的抽象数据类型，即堆栈、队列和双端队列，只允许在序列的一端或另一端进行更新操作。

我们希望有一个更一般的抽象。

位置列表 ADT(The Positional List ADT)

例

虽然我们将队列的 FIFO 准则作为等待与客户服务代表交谈的客户或排队购买演出门票的球迷的模型，但是队列 ADT 限制太强了。

我们希望设计一个抽象数据类型，它为用户提供了一种方法来引用序列中任意位置的元素，并执行任意的插入和删除。

位置列表 ADT(The Positional List ADT)

例

虽然我们将队列的 FIFO 准则作为等待与客户服务代表交谈的客户或排队购买演出门票的球迷的模型，但是队列 ADT 限制太强了。

如果等待的客户在到达客户服务队列前面之前决定挂断电话怎么办？或者，如果排队买票的人允许朋友在那个位置“插队”呢？

位置列表 ADT(The Positional List ADT)

例

虽然我们将队列的 FIFO 准则作为等待与客户服务代表交谈的客户或排队购买演出门票的球迷的模型，但是队列 ADT 限制太强了。

如果等待的客户在到达客户服务队列前面之前决定挂断电话怎么办？或者，如果排队买票的人允许朋友在那个位置“插队”呢？

我们希望设计一个抽象数据类型，它为用户提供了一种方法来引用序列中任意位置的元素，并执行任意的插入和删除。

位置列表 ADT(The Positional List ADT)

当使用基于数组的序列时，整数索引提供了一种很好的方法来描述元素的位置，或者应该进行插入或删除的位置。

位置列表 ADT(The Positional List ADT)

当使用基于数组的序列时，整数索引提供了一种很好的方法来描述元素的位置，或者应该进行插入或删除的位置。

然而，数字索引对于描述链表中的位置不是一个好的选择，因为我们无法有效地访问只知道其索引的条目；在链表的给定索引处查找元素需要从列表的开始或结束处以增量方式遍历该列表，并在遍历时对元素进行计数。

位置列表 ADT(The Positional List ADT)

此外，索引对于描述某些应用程序中的本地位置不是一个很好的抽象，因为在序列的前面发生的插入或删除会导致条目索引随时间变化。

例

准确地知道排队等候的人离队伍的前面有多远，可能不方便描述排队等候的人的位置。

位置列表 ADT(The Positional List ADT)

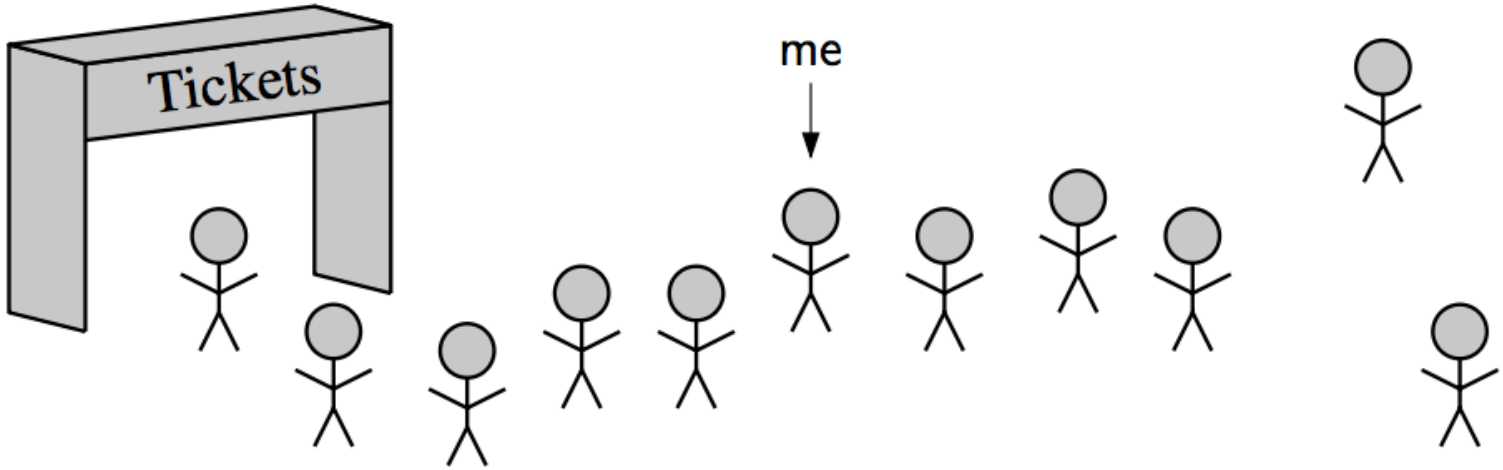


Figure 7.14: We wish to be able to identify the position of an element in a sequence without the use of an integer index.

位置列表 ADT(The Positional List ADT)

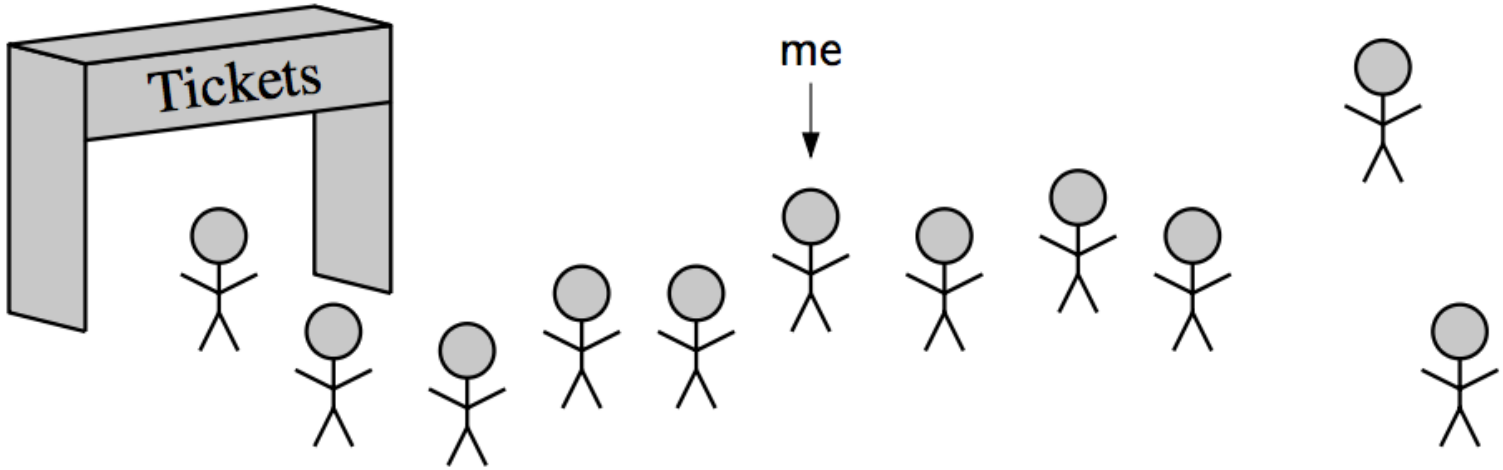


Figure 7.14: We wish to be able to identify the position of an element in a sequence without the use of an integer index.

我们希望对这样的情况进行建模，例如当一个已标识的人在到达前面之前离开时，或者当一个新的人被添加到紧跟在另一个已标识的人后面时。

位置列表 ADT(The Positional List ADT)

例

文本文档可以看作一个长字符序列。字处理程序使用一个cursor的抽象来描述文档中的一个位置，而不显式地使用整数索引，允许诸如“删除光标处的字符”或“在光标后插入新字符”之类的操作。

位置列表 ADT(The Positional List ADT)

链表结构的一个巨大好处是，只要我们被赋予对列表中相关节点的引用，就可以在列表的任意位置执行常数时间的插入和删除操作。

位置列表 ADT(The Positional List ADT)

链表结构的一个巨大好处是，只要我们被赋予对列表中相关节点的引用，就可以在列表的任意位置执行常数时间的插入和删除操作。

因此，开发一个 ADT 是非常诱人的，其中一个节点引用充当描述位置的机制。

位置列表 ADT(The Positional List ADT)

位置列表抽象数据类型

位置列表抽象数据类型

为了提供元素序列的一般抽象，并能够确定元素的位置，我们定义一个位置列表 ADT，以及一个位置 ADT来描述列表中的位置。

- 位置在位置列表中充当标记.
- 位置 p 不受列表中其他位置更改的影响；位置无效的唯一方式是发出显式命令删除它。

位置列表抽象数据类型

位置实例是一个简单的对象，仅支持以下方法：

- $p.element()$：返回存储在 p 位置的元素。

位置列表抽象数据类型

位置实例是一个简单的对象，仅支持以下方法：

- $p.element()$：返回存储在 p 位置的元素。

在位置列表 ADT 中，位置通常作为某些方法的参数和返回值。

位置列表抽象数据类型

位置列表的访问器方法有：

- *L.first()*：返回 *L* 的第一个元素的位置，如果 *L* 为空，则返回 *None*。
- *L.last()*：返回 *L* 的最后一个元素的位置，如果 *L* 为空，则返回 *None*。
- *L.before(p)*：返回 *L* 中 *p* 位置之前的位置，如果 *p* 为第一个位置，则返回 *None*。
- *L.after(p)*：返回 *L* 中 *p* 位置之后的位置，如果 *p* 为最后一个位置，则返回 *None*。
- *L.is_empty()*：如果 *L* 中不包含任何元素，则返回 *True*。
- *len(L)*：返回 *L* 中元素的个数。
- *iter(L)*：返回列表元素的前向迭代器。

位置列表抽象数据类型

位置列表的更新方法有

- $L.add\_first(e)$: 在 L 的表头处插入一个新元素 e，返回新元素的位置。
- $L.add\_last(e)$: 在 L 的表尾处插入一个新元素 e，返回新元素的位置。
- $L.add\_before(p, e)$: 在 L 中的位置 p 之前插入一个新元素 e，返回新元素的位置。
- $L.add\_after(p, e)$: 在 L 中的位置 p 之后插入一个新元素 e，返回新元素的位置。
- $L.replace(p, e)$: 用元素 e 替换位置 p 处的元素，返回位置 p 处的原元素。
- $L.delete(p)$: 删除并返回 L 中位置 p 处的元素，并使该位置失效。

位置列表抽象数据类型

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	$9_s, 7_p, 3_r$

位置列表 ADT(The Positional List ADT)

使用双重链表实现位置列表

使用双重链表实现位置列表

现在给出一个位置列表类的完整实现，该类使用双重链表。

性质

使用双重链表实现位置列表时，每个方法在最坏情况下都是常数时间操作。

使用双重链表实现位置列表

```
from doubly_linked_base import _DoublyLinkedBase
class PositionalList(_DoublyLinkedBase):
  class Position:
    def __init__(self, container, node):
      self._container = container
      self._node = node
    def element(self):
      return self._node._element
    def __eq__(self, other):
      return type(other) is type(self) and other
      ._node is self._node
    def __ne__(self, other):
      return not (self == other)
```

使用双重链表实现位置列表

```
def _validate(self, p):
  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:
    raise ValueError('p is no longer valid')
  return p._node
```

使用双重链表实现位置列表

```
def _make_position(self, node):
  if node is self._header or node is self.
  _trailer:
    return None
  else:
    return self.Position(self, node)
```

使用双重链表实现位置列表

```
  def first(self):
    return self._make_position(self._header.
    _next)
  def last(self):
    return self._make_position(self._trailer.
    _prev)
  def before(self, p):
    node = self._validate(p)
    return self._make_position(node._prev)
  def after(self, p):
    node = self._validate(p)
    return self._make_position(node._next)
```

使用双重链表实现位置列表

```
def __iter__(self):
  cursor = self.first()
  while cursor is not None:
    yield cursor.element()
    cursor = self.after(cursor)
def _insert_between(self, e, predecessor,
successor):
  node = super()._insert_between(e,
  predecessor, successor)
  return self._make_position(node)
```

使用双重链表实现位置列表

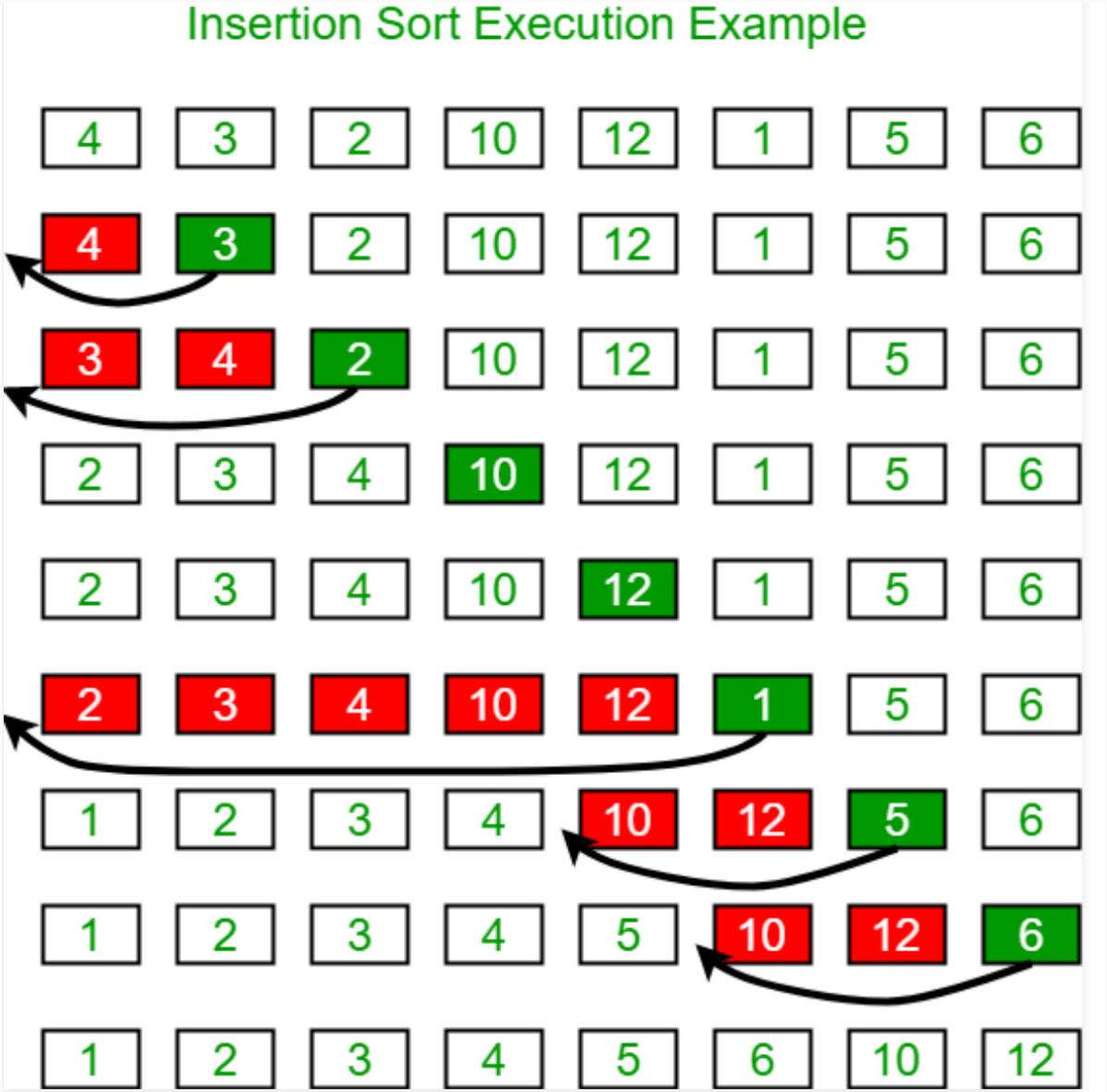
```
def add_first(self, e):
  return self._insert_between(e, self._header,
   self._header._next)
def add_last(self, e):
  return self._insert_between(e, self._trailer
  ._prev, self._trailer)
def add_before(self, p, e):
  original = self._validate(p)
  return self._insert_between(e, original.
  _prev, original)
def add_after(self, p, e):
  original = self._validate(p)
  return self._insert_between(e, original,
  original._next)
```

使用双重链表实现位置列表

```
  def delete(self, p):
    original = self._validate(p)
    return self._delete_node(original)
  def replace(self, p, e):
    original = self._validate(p)
    old_value = original._element
    original._element = e
    return old_value
```

对位置列表排序

对位置列表排序


Insertion Sort Execution Example
4 3 2 10 12 1 5 6
4 3 2 10 12 1 5 6
3 4 2 10 12 1 5 6
2 3 4 10 12 1 5 6
2 3 4 10 12 1 5 6
2 3 4 10 12 1 5 6
1 2 3 4 10 12 5 6
1 2 3 4 5 10 12 6
1 2 3 4 5 6 10 12

对位置列表排序

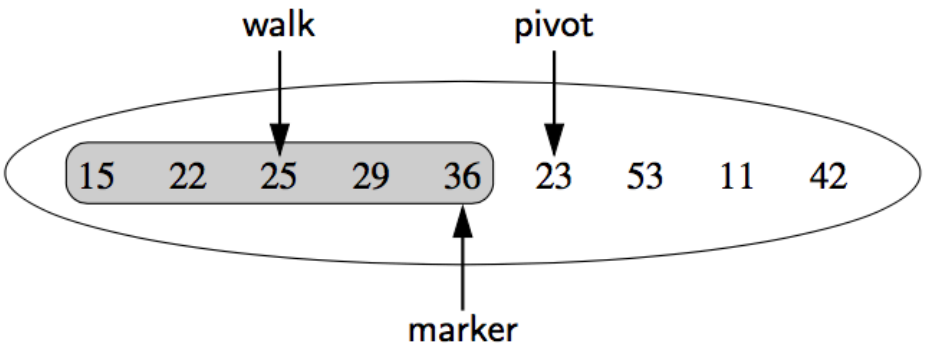


Figure 7.15: Overview of one step of our insertion-sort algorithm. The shaded elements, those up to and including marker, have already been sorted. In this step, the pivot's element should be relocated immediately before the walk position.

对位置列表排序

```
def insertion_sort(L):
  if len(L) > 1:
    marker = L.first()
    while marker != L.last():
      pivot = L.after(marker)
      value = pivot.element()
      if value > marker.element():
        marker = pivot
      else:
        walk = marker
        while walk != L.first() and L.before(
        walk).element() > value:
          walk = L.before(walk)
        L.delete(pivot)
        L.add_before(walk, value)
```