

云南大学数学与统计学院
上机实践报告

课程名称：数据结构与算法实验	年级：2015 级	上机实践成绩：
指导教师：陆正福	姓名：刘鹏	
上机实践名称：链表与表的抽象化实验	学号：20151910042	上机实践日期：2017-06-14
上机实践编号：No.07	组号：	上机实践时间：上午 3、4 节

一、实验目的

1. 熟悉链表以及表的抽象化等有关的概念，理解从数组到各类链表的选择；
2. 熟悉主讲教材 Chapter 7 的代码片段。

二、实验内容

1. 调试主讲教材 Chapter 7 的 Python 程序。

三、实验平台

Windows 10 1703 Enterprise 中文版；
Python 3.6.0；
Wing IDE Professional 6.0.4-1 集成开发环境。

四、实验记录与实验结果分析

1 题

用 Python 的链表，实现栈这一逻辑数据结构。

程序代码：

```
1  # 7.1.1 Implementing a Stack with a Singly Linked List
2
3  class Empty(Exception):
4      """Error attempting to access an element from an empty container."""
5      pass
6
7  class LinkedStack:
8      """LIFO Stack implementation using a singly linked list for storage."""
9
10     #----- nested _Node class -----
11     class _Node:
12         """Lightweight, nonpublic class for storing a singly linked node."""
13         __slots__ = '_element', '_next'      # streamline memory usage
14
15         def __init__(self, element, next):    # initialize node's fields
16             self._element = element          # reference to user's element
17             self._next = next                 # reference to next node
18
19     #----- stack methods -----
20     def __init__(self):
21         """Create an empty stack."""
22         self._head = None                    # reference to the head node
23         self._size = 0                       # number of stack elements
24
```

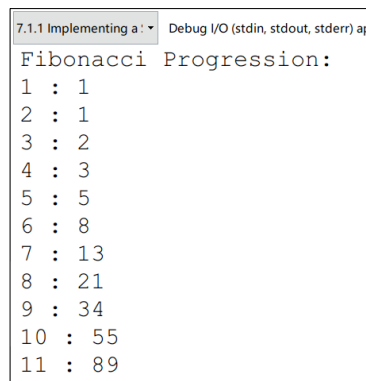
```

25     def __len__(self):
26         """Return the number of elements in the stack."""
27         return self._size
28
29     def is_empty(self):
30         """Return True if the stack is empty."""
31         return self._size == 0
32
33     def push(self,e):
34         """Add element e to the top of the stack."""
35         self._head = self._Node(e,self._head)    # create and link a new node
36         self._size += 1
37
38     def top(self):
39         """Return (but not remove) the element at the top of the stack.
40
41         Raise Empty exception if the stack is empty.
42         """
43         if self.is_empty():
44             raise Empty('Stack is empty')
45         return self._head._element                # top of stack is at head of list
46
47     def pop(self):
48         """Remove and return the element from the top of the stack (i.e., LIFO).
49
50         Raise Empty exception if the stack is empty.
51         """
52         if self.is_empty():
53             raise Empty('Stack is empty')
54         answer = self._head._element
55         self._head = self._head._next              # bypass the former top node
56         self._size -= 1
57         return answer
58
59     #----- my main function -----
60
61     a = LinkedStack()
62     b = LinkedStack()
63     f_1 = 1
64     f_2 = 1
65     a.push(1)
66     print('Fibonacci Progression:')
67     for i in range(11):
68         a.push(f_1)
69         f_1,f_2 = f_2,f_1 + f_2
70     for i in range(11):
71         b.push(a.pop())
72     for i in range(11):
73         print(i + 1, ': ',b.pop())

```

程序代码 1

运行结果:



The screenshot shows a debugger window titled "7.1.1 Implementing a" with a dropdown menu set to "Debug I/O (stdin, stdout, stderr) ap". The output text is as follows:

```
Fibonacci Progression:  
1 : 1  
2 : 1  
3 : 2  
4 : 3  
5 : 5  
6 : 8  
7 : 13  
8 : 21  
9 : 34  
10 : 55  
11 : 89
```

运行结果 1

代码分析:

由于进行了封装，所以就不再分析如何使用了，ArrayStack 与 LinkedStack 的 interface 基本一样，但是操作的时间复杂度不一样。

由于嵌套类的样式早就谈过，所以这里也没有新意。

五、教材翻译

Translation

Chapter 7 Linked Lists

* 第七章 链表

In Chapter 5 we carefully examined Python's array-based list class, and in Chapter 6 we demonstrated use of that class in implementing the classic stack, queue, and dequeue ADTs. Python's list class is highly optimized, and often a great choice for storage. With that said, there are some notable disadvantages:

* 我们在第五章详细讨论了 Python 下面的基于数组的列表类，后来又在第六章里面，通过经典的栈、队列、双端队列等抽象数据类型的程序实现，展示了列表类的用途。Python 的列表类非常高效，通常是用来存储的良好选择。但是，列表类也有很多劣势，如下：

1. The length of a dynamic array might be longer than the actual number of elements that it stores.
* 动态数组所占用的实际内存可能比数组本身的理论内存要大。
2. Amortized bounds for operations may be unacceptable in real-time systems.
* 在实时系统中，均摊操作可能是不可接受的。
3. Insertions and deletions at interior positions of an array are expensive.
* 在内部位置插入或者删除一个元素的代价可能很大。

In this chapter, we introduce a data structure known as a **linked list**, which provides an alternative to an array-based sequence (such as a Python list). Both array-based sequences

and linked lists keep elements in a certain order, but using a very different style. An array provides the more centralized representation, with one large chunk of memory capable of accommodating references to many elements. A linked list, in contrast, relies on a more distributed representation in which a lightweight object, known as a **node**, is allocated for each element. Each node maintains a reference to its element and one or more references to neighboring nodes in order to collectively represent the linear order of the sequence.

* 我们将在这一章中，介绍一种叫做链表的数据结构，这种结构可以在某些时候替代基于数组的序列。无论是基于数组的序列还是链表，他们的元素都是按照一定的顺序进行保存的，但是两者的保存方式却大有不同。数组采用的是一种集中性很高的实现方法，那就是开辟一大块连续的内存区域用以存放元素。而链表则不然，链表采用一种分布式的实现方式，而这个方式是依靠为每一个元素分配一个叫做节点的轻量级对象而实现的。每一个节点都包含着与它的对应元素所匹配的指向，同时也包含着其相邻的节点的指向，这样一来就表现出了序列的线性结构。

We will demonstrate a trade-off of advantages and disadvantages when contrasting array-based sequences and linked lists. Elements of a linked list cannot be efficiently accessed by a numeric index k , and we cannot tell just by examining a node if it is the second, fifth, or twentieth node in the list. However, linked lists avoid the three disadvantages noted above for array-based sequence.

* 当对比基于数组的数列与链表的时候，我们会均衡地展示两者的优缺点。链表中的元素不能通过下标索引进行高效的访问，并且我们不能通过检查节点来判断一个节点在表中的位置。然而，链表可以很好地避免上面提到的那三个数组序列的弊端。

7.1 Singly Linked List

* 单链表

A singly linked list, in its simplest form, is a collection of nodes that collectively form a linear sequence. Each node stores a reference to an object that is an element of the sequence, as well as a reference to the next node of the list (see Figure 7.1 and 7.2)

* 单链表的最简单形式就是将节点链接起来，构成一个线性序列。每个节点存储序列元素的对象地址，同时存储下一个节点的地址（参见图 7.1 和 7.2）

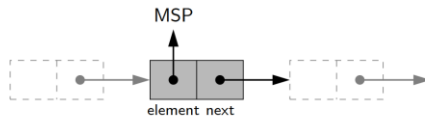


Figure 7.1: Example of a node instance that forms part of a singly linked list. The node's element member references an arbitrary object that is an element of the sequence (the airport code MSP, in this example), while the next member references the subsequent node of the linked list (or None if there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

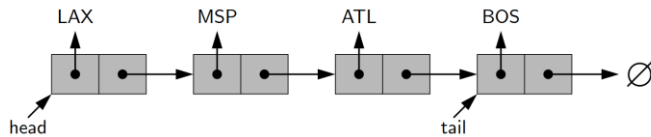


Figure 7.2: Example of a singly linked list whose elements are strings indicating airport codes. The list instance maintains a member named head that identifies the first node of the list, and in some applications another member named tail that identifies the last node of the list. The None object is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

The first and last node of a linked list are known as the **head** and **tail** of the list, respectively. By starting at the head, and moving from one node to another by following each node's next reference, we can reach the tail of the list. We can identify the tail as the node having None as its next reference. This process is commonly known as **traversing** the linked list. Because the next reference of a node can be viewed as a **link** or **pointer** to another node, the process of traversing a list is also known as **link hopping** or **pointer hopping**.

* 链表的第一个和最后一个节点分别被称为列表的头和尾。从 head 开始，通过跟随每个节点的下一个指向，可以从一个节点移动到另一个节点，我们可以到达列表的尾部。我们可以将尾部的下一个指向认作 None。这个过程通常称为链表遍历。因为节点中所包含的下一个引用可被视为到另一个节点的链接或指针，所以遍历列表的过程也被称为链路跳变或指针跳跃。

A linked list's representation in memory relies on the collaboration of many objects. Each node is represented as a unique object, with that instance storing a reference to its element and a reference to the next node (or None). Another object represents the linked list as a whole. Minimally, the linked list instance must keep a reference to the head of the list. Without an explicit reference to the head, there would be no way to locate that node (or indirectly, any others). There is not an absolute need to store a direct reference to the tail of the list, as it could otherwise be located by starting at the head and traversing the rest of the list.

However, storing an explicit reference to the tail node is a common convenience to avoid such a traversal. In similar regard, it is common for the linked list instance to keep a count of the total number of nodes that comprise the list (commonly described as the **size** of the list), to avoid the need to traverse the list to count the nodes.

* 链表在内存中的形成依赖于许多对象的协作。每个节点都表示一个对象对象，这个存储存储着数据元素的地址和下一个节点（或 None）的地址。另一个对象表示整个链表。最小的链表必须保留对列表头部的引用。没有明确提及头，就没有办法找到那个节点。千万不需要存储对列表尾部的直接引用，否则可能会从头开始并遍历列表的其余部分。然而，存储对尾部节点的显式引用是避免这种遍历的常见方便。类似地，链表示例通常保留构成列表的节点总数（通常被描述为列表的大小），以免需要遍历列表来计数节点。

For the remainder of this chapter, we continue to illustrate nodes as objects, and each node's "next" reference as a pointer. However, for the sake of simplicity, we illustrate a node's element embedded directly within the node structure, even though the element is, in fact, an independent object. For example, Figure 7.3 is a more compact illustration of the linked list from Figure 7.2.

* 对于本章的其余部分，我们继续将节点描述为对象，并将每个节点的“下一个”引用作为指针。然而，为了简单起见，我们把元素直接嵌入到了节点里，尽管该元素实际上是一个独立的对象。例如，图 7.3 是来自图 7.2 的链表的更紧凑的说明。

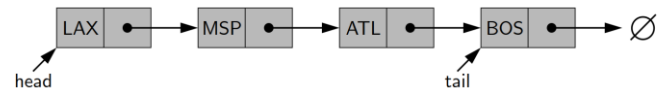


Figure 7.3: A compact illustration of a singly linked list, with elements embedded in the nodes (rather than more accurately drawn as references to external objects).

An important property of a linked list is that it does not have a predetermined fixed size; it uses space proportionally to its current number of elements. When using a singly linked list, we can easily insert an element at the head of the list, as shown in Figure 7.4, and described with pseudo-code in Code Fragment 7.1. The main idea is that we create a new node, set its element to the new element, set its next link to refer to the current head, and then set the list's head to point to the new node.

* 链表的一个重要属性是它没有预定的固定大小；它使用空间与其当前数量的元素成比例。当使用单链表时，我们可以轻松地在列表的头部插入一个元素，如图 7.4 所示，并用 Code Fragment 7.1 中的伪代码进行描述。主要思想是我们创建一个新节点，将其元素设置为新元素，将其下一个链接设置为指向当前头，然后将列表的头设置为指向新节点。

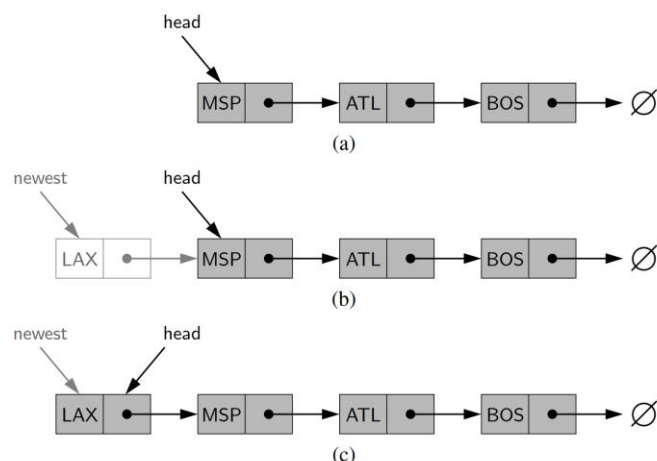


Figure 7.4: Insertion of an element at the head of a singly linked list: (a) before the insertion; (b) after creation of a new node; (c) after reassignment of the head reference.

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.

Inserting an Element at the Tail of a Singly Linked List

* 在单链接列表的尾部插入元素

We can also easily insert an element at the tail of the list, provided we keep a reference to the tail node, as shown in Figure 7.5. In this case, we create a new node, assign its next reference to None, set the next reference of the tail to point to this new node, and then update the tail reference itself to this new node. We give the details in Code Fragment 7.2.

* 如果我们保留对尾节点的引用，我们也可以轻松地在列表的尾部插入一个元素，如图 7.5 所示。在这种情况下，我们创建一个新节点，将其下一个引用设置为 None，设置原来尾部的下一个引用为该新节点，然后将尾部更新为此新节点。我们在 Code Fragment 7.2 中给出细节。

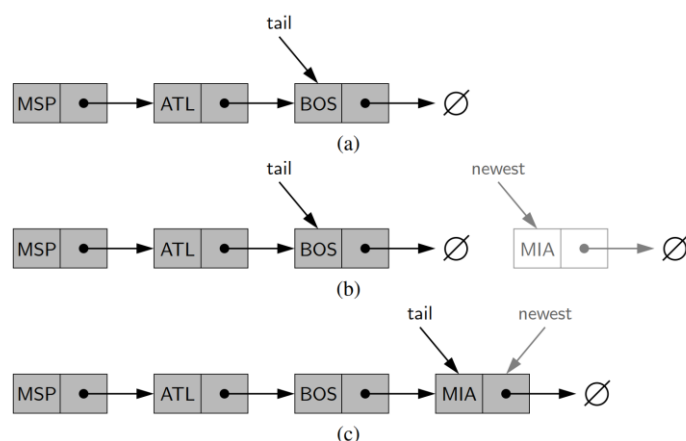


Figure 7.5: Insertion at the tail of a singly linked list: (a) before the insertion; (b) after creation of a new node; (c) after reassignment of the tail reference. Note that we must set the next link of the tail in (b) before we assign the tail variable to point to the new node in (c).

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 an Element from a Singly Linked List

* 从单独链接的列表中删除元素

Removing an element from the *head* of a singly linked list is essentially the reverse operation of inserting a new element at the head. This operation is illustrated in Figure 7.6 and given in detail in Code Fragment 7.3.

* 从单链表的头部移除元素本质上是在头部插入新元素的相反操作。该操作在图 7.6 中示出，并在代码片段 7.3 中给出。

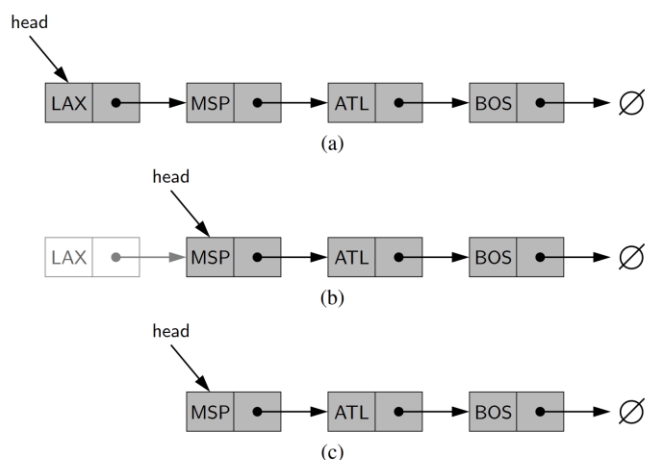


Figure 7.6: Removal of an element at the head of a singly linked list: (a) before the removal; (b) after "linking out" the old head; (c) final configuration.

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).

* 不幸的是，我们不能轻易删除单链表的最后一个节点。即使我们直接存储列表最后一个节点的尾部引用，我们必须能够在最后一个节点之前访问节点，以便删除最后一个节点。但是我们不能通过尾部的下一个链接到尾部之前的节点。访问此节点的唯一方法是从列表的头部开始，并搜索列表中的所有方法。但是这样的一连串跳频操作可能需要很长时间。如果我们想要有效地支持这样一个操作，我们将需要使我们的列表双重关联（如我们在 7.3 节中所述）。

7.1.1 Implementing a Stack with a Singly Linked

* 7.1.1 节 用单链表实现栈逻辑结构

In this section, we demonstrate use of a singly linked list by providing a complete Python implementation of the stack ADT (see Section 6.1). In designing such an implementation, we need to decide whether to model the top of the stack at the head or at the tail of the list. There is clearly a best choice here; we can efficiently insert and delete elements in constant time only at the head. Since all stack operations affect the top, we orient the top of the stack at the head of our list.

* 在本节中，我们通过提供堆栈 ADT 的完整 Python 实现来演示单链表的使用（见第 6.1 节）。在设计这样的一个实现时，我们需要决定把栈结构的顶部放在 `head` 处还是 `tail` 处。由于所有堆栈操作都会影响顶部，因此我们将堆栈的顶部定位在列表的头部。这是个很好的选择；通过这个选择，我们可以有效地插入和删除元素，而时间复杂度为常量级。

To represent individual nodes of the list, we develop a lightweight `_Node` class. This class will never be directly exposed to the user of our stack class, so we will formally define it as a non-public, nested class of our eventual `LinkedStack` class (see Section 2.5.1 for discussion of nested classes). The definition of the `_Node` class is shown in Code Fragment 7.4.

* 为了表示列表的各个节点，我们开发了一个轻量级的 `_Node` 类。这个类永远不会直接暴露给我们的栈用户，所以我们将正式定义我们最终的 `LinkedStack` 类的非公开的、嵌套的类（有关嵌套类的讨论，请参见第 2.5.1 节）。节点类的定义如代码片段 7.4 所示。

A node has only two instance variables: `_element` and `_next`. We intentionally define `__slots__` to streamline the memory usage (see page 99 of Section 2.5.1 for discussion), because there may potentially be many node instances in a single list. The constructor of the `_Node` class is designed for our convenience, allowing us to specify initial values for both fields of a newly created node.

* 节点只有两个实例变量：`_element` 和 `_next`。我们有意定义 `__slots__` 来简化内存使用（参见第 2.5.1 节的讨论），因为单个列表中可能有许多节点实例。`_Node` 类的构造函数是为了方便起见而设计的，允许我们为新创建的节点的两个字段指定初始值。

A complete implementation of our `LinkedStack` class is given in Code Fragments 7.5 and 7.6. Each stack instance maintains two variables. The `head` member is a reference to the node at the head of the list (or `None`, if the stack is empty). We keep track of the current number of elements with the `_size` instance variable, for otherwise we would be forced to traverse the entire list to count the number of elements when reporting the size of the stack.

* 在 Code Fragments 7.5 和 7.6 中给出了 `LinkedStack` 类的完整实现。每个堆栈实例维护两个变量。头成员是对列表

头部的节点的引用（如果堆栈为空，则为 `None`）。我们使用 `_size` 变量跟踪当前元素的数量，否则我们将不得不通过遍历整个列表来计算栈的元素数量。

The implementation of `push` essentially mirrors the pseudo-code for insertion at the head of a singly linked list as outlined in Code Fragment 7.1. When we push a new element `e` onto the stack, we accomplish the necessary changes to the linked structure by invoking the constructor of the `_Node` class as follows:

* 按照代码片段 7.1 中所述，`push` 的实现基本上反映了用于插入到单链表的头部的伪代码。当我们将新的元素 `e` 推入栈时，我们通过调用 `_Node` 类的构造函数来完成对链接结构的必要更改，如下所示：

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

Note that the `next` field of the new node is set to the *existing* top node, and then `self.head` is reassigned to the new node.

* 请注意，新节点的下一个节点是老的节点。头被重新分配给新节点。

When implementing the `top` method, the goal is to return the *element* that is at the top of the stack. When the stack is empty, we raise an `Empty` exception, as originally defined in Code Fragment 6.1 of Chapter 6. When the stack is nonempty, `self._head` is a reference to the first *node* of the linked list. The top element can be identified as `self._head._element`.

* 当实现顶级方法时，目标是重新转换堆栈顶部的元素。当堆栈为空时，我们引发一个 `Empty` 异常，如第 6 章的 Code Fragment 6.1 中最初定义的那样。当堆栈非空时，`self._head` 是对链表第一个节点的引用。顶部元素可以被识别为 `self._head._element`。

Our implementation of `pop` essentially mirrors the pseudo-code given in Code Fragment 7.3, except that we maintain a local reference to the element that is stored at the node that is being removed, and we return that element to the caller of `pop`.

* 我们的 `pop` 的实现本质上反映了代码片段 7.3 中给出的伪代码，除了我们存储了对被删除的节点上存储的元素的本地引用，并将该元素返回给 `pop` 的调用者。

The analysis of our `LinkedStack` operations is given in Table 7.1. We see that all of the methods complete in *worst-case* constant time. This is in contrast to the amortized bounds for the `ArrayStack` that were given in Table 6.2.

* 我们的 `LinkedStack` 操作的分析在表 7.1 中给出。我们看到所有的方法在最坏的情况下都是在常量时间内完成的。这与表 6.2 中给出的 `ArrayStack` 的异构化边界形成对比。

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.

7.1.2 Implementing a Queue with a Singly Linked List

*

As we did for the stack ADT, we can use a singly linked list to implement the queue ADT while supporting worst-case $O(1)$ -time for all operations. Because we need to perform operations on both ends of the queue, we will explicitly maintain both a head reference and a tail reference as instance variables for each queue. The natural orientation for a queue is to align the front of the queue with the head of the list, and the back of the queue with the tail of the list, because we must be able to enqueue elements at the back, and dequeue them from the front. (Recall from the introduction of Section 7.1 that we are unable to efficiently remove elements from the tail of a singly linked list.) Our implementation of a `LinkedList` class is given in Code Fragments 7.7 and 7.8.

*

Many aspects of our implementation are similar to that of the `LinkedList` class, such as the definition of the nested `Node` class. Our implementation of `dequeue` for `LinkedList` is similar to that of `pop` for `LinkedList`, as both remove the head of the linked list. However, there is a subtle difference because our

queue must accurately maintain the tail reference (no such variable was maintained for our stack). In general, an operation at the head has no effect on the tail, but when `dequeue` is invoked on a queue with one element, we are simultaneously removing the tail of the list. We therefore set `self.tail` to `None` for consistency.

*

There is a similar complication in our implementation of `enqueue`. The newest node always becomes the new tail. Yet a distinction is made depending on whether that new node is the only node in the list. In that case, it also becomes the new head; otherwise the new node must be linked immediately after the existing tail node.

*

In terms of performance, the `LinkedList` is similar to the `LinkedList` in that all operations run in worst-case constant time, and the space usage is linear in the current number of elements.

*

7.2 Circularly Linked Lists

In Section 6.2.2, we introduced the notion of a “circular” array and demonstrated its use in implementing the queue ADT. In reality, the notion of a circular array was artificial, in that there was nothing about the representation of the array itself that was circular in structure. It was our use of modular arithmetic when “advancing” an index from the last slot to the first slot that provided such an abstraction.

In the case of linked lists, there is a more tangible notion of a circularly linked list, as we can have the tail of the list use its next reference to point back to the head of the list, as shown in Figure 7.7. We call such a structure a **circularly linked list**.

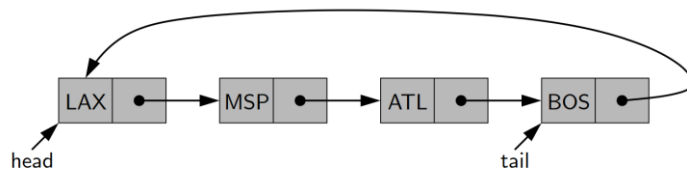


Figure 7.7: Example of a singly linked list with circular structure.

A circularly linked list provides a more general model than a standard linked list for data sets that are cyclic, that is, which do not have any particular notion of a beginning and end. Figure 7.8 provides a more symmetric illustration of the same circular list structure as Figure 7.7.

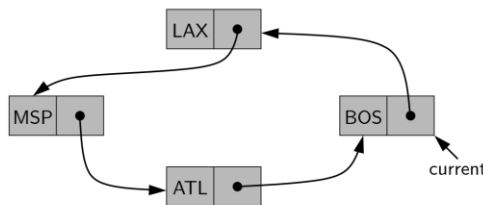


Figure 7.8: Example of a circular linked list, with current denoting a reference to a select node.

A circular view similar to Figure 7.8 could be used, for example, to describe the order of train stops in the Chicago loop, or the order in which players take turns during a game. Even though a circularly linked list has no beginning or end, per se, we must maintain a reference to a particular node in order to make use of the list. We use the identifier **current** to describe such a designated node. By setting $current = current.next$, we can effectively advance through the nodes of the list.

7.2.1 Round-Robin Schedulers

To motivate the use of a circularly linked list, we consider a **round-robin** scheduler, which iterates through a collection of elements in a circular fashion and “services” each element by performing a given action on it. Such a scheduler is used, for example, to fairly allocate a resource that must be shared by a collection of clients. For instance, round-robin scheduling is often used to allocate slices of CPU time to various applications running concurrently on a computer.

A round-robin scheduler could be implemented with the general queue ADT, by repeatedly performing the following steps on queue Q (see Figure 7.9):

1. $e = Q.dequeue()$
2. Service element e
3. $Q.enqueue(e)$

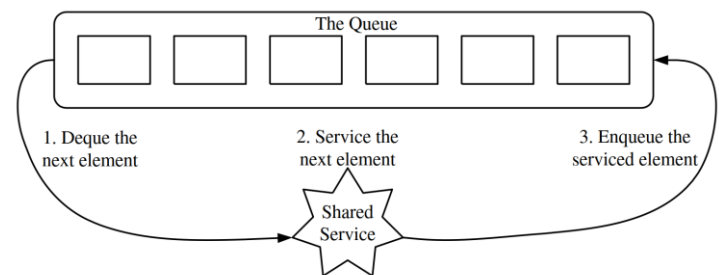


Figure 7.9: The three iterative steps for round-robin scheduling using a queue.

If we use of the `LinkedList` class of Section 7.1.2 for such an application, there is unnecessary effort in the combination of a dequeue operation followed soon after by an enqueue of the same element. One node is removed from the list, with appropriate adjustments to the head of the list and the size decremented, and then a new node is created to reinsert at the tail of the list and the size is incremented.

If using a circularly linked list, the effective transfer of an item from the “head” of the list to the “tail” of the list can be accomplished by advancing a reference that marks the boundary of the queue. We will next provide an implementation of a `CircularQueue` class that supports the entire queue ADT, together with an additional method, `rotate()`, that moves the first element of the queue to the back. (A similar method is supported by the deque class of Python’s `collections` module; see Table 6.4.) With this operation, a round-robin schedule can more efficiently be implemented by repeatedly performing the following steps:

1. Service element $Q.front()$
2. $Q.rotate()$

7.2.2 Implementing a Queue with a Circularly Linked List

To implement the queue ADT using a circularly linked list, we rely on the intuition of Figure 7.7, in which the queue has a head and a tail, but with the next reference of the tail linked to the head. Given such a model, there is no need for us to explicitly store references to both the head and the tail; as long as we keep a reference to the tail, we can always find the head by following the tail's next reference.

Code Fragments 7.9 and 7.10 provide an implementation of a `CircularQueue` class based on this model. The only two instance variables are `_tail`, which is a reference to the tail node (or `None` when empty), and `_size`, which is the current number of

elements in the queue. When an operation involves the front of the queue, we recognize `self._tail._next` as the head of the queue. When `enqueue` is called, a new node is placed just after the tail but before the current head, and then the new node becomes the tail.

In addition to the traditional queue operations, the `CircularQueue` class supports a `rotate` method that more efficiently enacts the combination of removing the front element and reinserting it at the back of the queue. With the circular representation, we simply set `self._tail = self._tail._next` to make the old head become the new tail (with the node after the old head becoming the new head).

7.3 Doubly Linked Lists

In a singly linked list, each node maintains a reference to the node that is immediately after it. We have demonstrated the usefulness of such a representation when managing a sequence of elements. However, there are limitations that stem from the asymmetry of a singly linked list. In the opening of Section 7.1, we emphasized that 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).

To provide greater symmetry, we define a linked list in which each node keeps an explicit reference to the node before it and a reference to the node after it. Such a structure is known as a **doubly linked list**. These lists allow a greater variety of $O(1)$ -time update operations, including insertions and deletions at arbitrary positions within the list. We continue to use the term “next” for the reference to the node that follows another, and we introduce the term “prev” for the reference to the node that precedes it.

Header and Trailer Sentinels

In order to avoid some special cases when operating near the boundaries of a doubly linked list, it helps to add special nodes at both ends of the list: a **header** node at the beginning of the list, and a **trailer** node at the end of the list. These “dummy” nodes are known as **sentinels** (or guards), and they do not store elements of the primary sequence. A doubly linked list with such sentinels is shown in Figure 7.10.

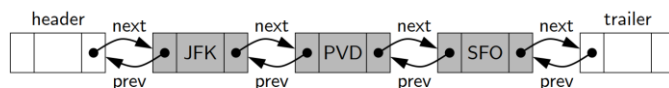


Figure 7.10: A doubly linked list representing the sequence { JFK, PVD, SFO }, using sentinels header and trailer to demarcate the ends of the list.

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

Advantage of Using Sentinels

Although we could implement a doubly linked list without sentinel nodes (as we did with our singly linked list in Section 7.1), the slight extra space devoted to the sentinels greatly simplifies the logic of our operations. Most notably, the header and trailer

nodes never change—only the nodes between them change. Furthermore, we can treat all insertions in a unified manner, because a new node will always be placed between a pair of existing nodes. In similar fashion, every element that is to be deleted is guaranteed to be stored in a node that has neighbors on each side.

For contrast, look back at our `LinkedList` implementation from Section 7.1.2. Its `enqueue` method, given in Code Fragment 7.8, adds a new node to the end of the list. However, its implementation required a conditional to manage the special case of inserting into an empty list. In the general case, the new node was linked after the existing tail. But when adding to an empty list, there is no existing tail; instead it is necessary to reassign self.head to reference the new node. The use of a sentinel node in that implementation would eliminate the special case, as there would always be an existing node (possibly the header) before a new node.

Inserting and Deleting with a Doubly Linked List

Every insertion into our doubly linked list representation will take place between a pair of existing nodes, as diagrammed in Figure 7.11. For example, when a new element is inserted at the front of the sequence, we will simply add the new node *between* the header and the node that is currently after the header. (See Figure 7.12.)

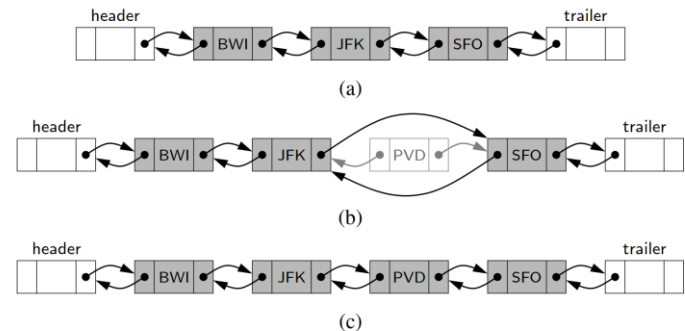


Figure 7.11: Adding an element to a doubly linked list with header and trailer sentinels: (a) before the operation; (b) after creating the new node; (c) after linking the neighbors to the new node.

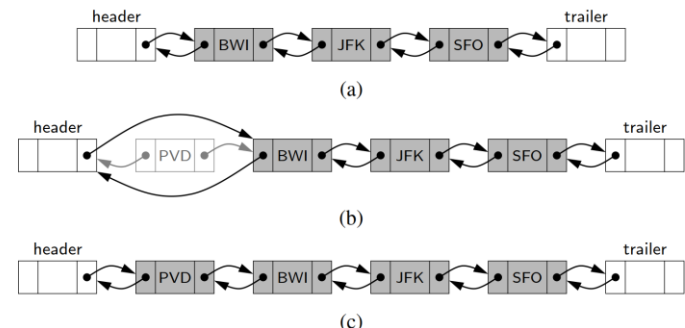


Figure 7.12: Adding an element to the front of a sequence represented by a doubly linked list with header and trailer sentinels: (a) before the operation; (b) after creating the new node; (c) after linking the neighbors to the new node.

The deletion of a node, portrayed in Figure 7.13, proceeds in

the opposite fashion of an insertion. The two neighbors of the node to be deleted are linked directly to each other, thereby bypassing the original node. As a result, that node will no longer be considered part of the list and it can be reclaimed by the system. Because of our use of sentinels, the same implementation can be used when deleting the first or the last element of a sequence, because even such an element will be stored at a node that lies between two others.

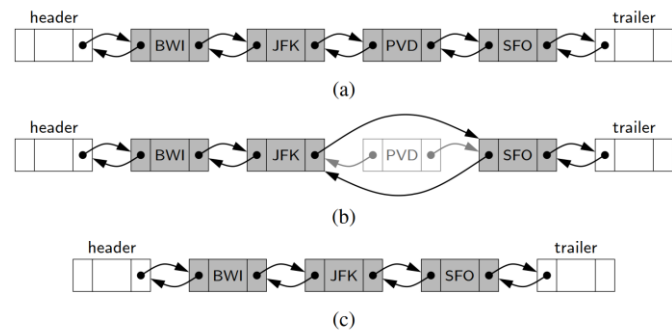


Figure 7.13: Removing the element PVD from a doubly linked list: (a) before the removal; (b) after linking out the old node; (c) after the removal (and garbage collection).

7.3.1 Basic Implementation of a Doubly Linked List

We begin by providing a preliminary implementation of a doubly linked list, in the form of a class named `DoublyLinkedListBase`. We intentionally name the class with a leading underscore because we do not intend for it to provide a coherent public interface for general use. We will see that linked lists can support general insertions and deletions in $O(1)$ worst-case time, but only if the location of an operation can be succinctly identified. With array-based sequences, an integer index was a convenient means for describing a position within a sequence. However, an index is not convenient for linked lists as there is no efficient way to find the j^{th} element; it would seem to require a traversal of a portion of the list.

When working with a linked list, the most direct way to describe the location of an operation is by identifying a relevant node of the list. However, we prefer to encapsulate the inner workings of our data structure to avoid having users directly access nodes of a list. In the remainder of this chapter, we will develop two public classes that inherit from our `DoublyLinkedListBase` class to provide more coherent abstractions. Specifically, in Section 7.3.2, we provide a `LinkedDeque` class that implements the double-ended queue ADT introduced in Section 6.3; that class only supports operations at the ends of the queue, so there is no need for a user to identify an interior position within the list. In Section 7.4, we introduce a new `PositionalList` abstraction that provides a public interface that allows arbitrary insertions and deletions from a list.

Our low-level `DoublyLinkedListBase` class relies on the use of a nonpublic `Node` class that is similar to that for a singly linked list, as given in Code Fragment 7.4, except that the doubly linked

version includes a `prev` attribute, in addition to the `next` and `element` attributes, as shown in Code Fragment 7.11.

The remainder of our `DoublyLinkedListBase` class is given in Code Fragment 7.12. The constructor instantiates the two sentinel nodes and links them directly to each other. We maintain a `size` member and provide public support for `len` and `is empty` so that these behaviors can be directly inherited by the subclasses.

The other two methods of our class are the nonpublic utilities, `insert between` and `delete node`. These provide generic support for insertions and deletions, respectively, but require one or more node references as parameters. The implementation of the `insert between` method is modeled upon the algorithm that was previously portrayed in Figure 7.11. It creates a new node, with that node's fields initialized to link to the specified neighboring nodes. Then the fields of the neighboring nodes are updated to include the newest node in the list. For later convenience, the method returns a reference to the newly created node.

The implementation of the `delete node` method is modeled upon the algorithm portrayed in Figure 7.13. The neighbors of the node to be deleted are linked directly to each other, thereby bypassing the deleted node from the list. As a formality, we intentionally reset the `prev`, `next`, and `element` fields of the deleted node to `None` (after recording the element to be returned). Although the deleted node will be ignored by the rest of the list, setting its fields to `None` is advantageous as it may help Python's garbage collection, since unnecessary links to the other nodes and the stored element are eliminated. We will also rely on this configuration to recognize a node as “deprecated” when it is no longer part of the list.

7.3.2 Implementing a Deque with a Doubly Linked List

The double-ended queue (deque) ADT was introduced in Section 6.3. With an array-based implementation, we achieve all operations in *amortized* $O(1)$ time, due to the occasional need to resize the array. With an implementation based upon a doubly linked list, we can achieve all deque operation in *worst-case* $O(1)$ time.

We provide an implementation of a `LinkedDeque` class (Code Fragment 7.13) that inherits from the `DoublyLinkedListBase` class of the preceding section. We do not provide an explicit `init` method for the `LinkedDeque` class, as the inherited version of that method suffices to initialize a new instance. We also rely on the inherited methods `len` and `is empty` in meeting the deque ADT.

With the use of sentinels, the key to our implementation is to

remember that the header does not store the first element of the deque—it is the node just *after* the header that stores the first element (assuming the deque is nonempty). Similarly, the node just *before* the trailer stores the last element of the deque.

We use the inherited `insert between` method to insert at either end of the deque. To insert an element at the front of the deque, we place it immediately between the header and the node just after the header. An insertion at the end of deque is placed immediately before the trailer node. Note that these operations succeed, even when the deque is empty; in such a situation, the new node is placed between the two sentinels. When deleting an element from a nonempty deque, we rely upon the inherited `delete node` method, knowing that the designated node is assured to have neighbors on each side.

7.4 The Positional List ADT

The abstract data types that we have considered thus far, namely stacks, queues, and double-ended queues, only allow update operations that occur at one end of a sequence or the other. We wish to have a more general abstraction. For example, although we motivated the FIFO semantics of a queue as a model for customers who are waiting to speak with a customer service representative, or fans who are waiting in line to buy tickets to a show, the queue ADT is too limiting. 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? We would like to design an abstract data type that provides a user a way to refer to elements anywhere in a sequence, and to perform arbitrary insertions and deletions.

When working with array-based sequences (such as a Python list), integer indices provide an excellent means for describing the location of an element, or the location at which an insertion or deletion should take place. However, numeric indices are not a good choice for describing positions within a linked list because we cannot efficiently access an entry knowing only its index; finding an element at a given index within a linked list requires traversing the list incrementally from its beginning or end, counting elements as we go.

Furthermore, 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. For example, it may not be convenient to describe the location of a person waiting in line by knowing precisely how far away that person is from the front of the line. We prefer an abstraction, as characterized in Figure 7.14, in which there is some other means for describing a position. We then wish to model situations such as when an identified person leaves the line before reaching the front, or in which a new person is added to a line immediately behind another identified person.

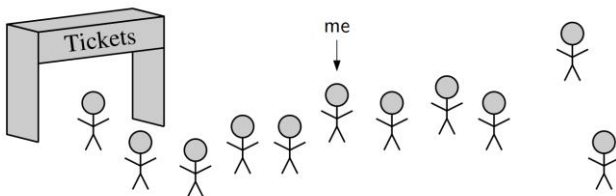


Figure 7.14: We wish to be able to identify the position of an element in a sequence without the use of an integer index.

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.

A Node Reference as a Position?

One of the great benefits of a linked list structure is that it is possible to perform $O(1)$ -time insertions and deletions at arbitrary positions of the list, as long as we are given a reference to a relevant node of the list. It is therefore very tempting to develop an ADT in which a node reference serves as the mechanism for describing a position. In fact, our `DoublyLinkedListBase` class of Section 7.3.1 has methods `insert between` and `delete node` that accept node references as parameters.

However, such direct use of nodes would violate the object-oriented design principles of abstraction and encapsulation that were introduced in Chapter 2. There are several reasons to prefer that we encapsulate the nodes of a linked list, for both our sake and for the benefit of users of our abstraction.

- It will be simpler for users of our data structure if they are not bothered with unnecessary details of our implementation, such as low-level manipulation of nodes, or our reliance on the use of sentinel nodes. Notice that to use the `insert between` method of our `DoublyLinkedListBase` class to add a node at the beginning of a sequence, the header sentinel must be sent as a parameter.
- We can provide a more robust data structure if we do not permit users to directly access or manipulate the nodes. In that way, we ensure that users cannot invalidate the consistency of a list by mismanaging the linking of nodes. A more subtle problem arises if a user were allowed to call the `insert between` or `delete node` method of our `DoublyLinkedListBase` class, sending a node that does not belong to the given list as a parameter. (Go back and look at that code and see why it causes a problem!)
- By better encapsulating the internal details of our implementation, we have greater flexibility to redesign the data structure and improve its performance. In fact, with a well-designed abstraction, we can provide a notion of a non-numeric position, even if using an array-based sequence.

For these reasons, instead of relying directly on nodes, we introduce an independent *position* abstraction to denote the location of an element within a list, and then a complete *positional list ADT* that can encapsulate a doubly linked list (or even an array-based sequence; see Exercise P-7.46).

7.4.1 The Positional List Abstract Data Type

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 as well as a simpler position abstract data type to describe a location within a list. 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:

$p.\text{element}()$: Return the element stored at position p .

In the context of the positional list ADT, positions serve as parameters to some methods and as return values from other methods. In describing the behaviors of a positional list, we begin by presenting the accessor methods supported by a list L :

$L.\text{first}()$: Return the position of the first element of L , or None if L is empty.

$L.\text{last}()$: Return the position of the last element of L , or None if L is empty.

$L.\text{before}(p)$: Return the position of L immediately before position p , or None if p is the first position.

$L.\text{after}(p)$: Return the position of L immediately after position p , or None if p is the last position.

$L.\text{is empty}()$: Return True if list L does not contain any elements.

$\text{len}(L)$: Return the number of elements in the list.

$\text{iter}(L)$: Return a forward iterator for the elements of the list. See Section 1.8 for discussion of iterators in Python.

The positional list ADT also includes the following update methods:

$L.\text{add_first}(e)$: Insert a new element e at the front of L , returning the position of the new element.

$L.\text{add_last}(e)$: Insert a new element e at the back of L , returning the position of the new element.

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

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

$L.\text{replace}(p, e)$: Replace the element at position p with element e , returning the element formerly at position p .

$L.\text{delete}(p)$: Remove and return the element at position p in L , invalidating the position.

For those methods of the ADT that accept a position p as a parameter, an error occurs if p is not a valid position for list L .

Note well that the $\text{first}()$ and $\text{last}()$ methods of the positional list ADT return the associated positions, not the elements. (This is in contrast to the corresponding first and last methods of the deque ADT.) The first element of a positional list can be determined by subsequently invoking the $\text{element}()$ method on that position, as $L.\text{first}().\text{element}()$. The advantage of receiving a position as a return value is that we can use that position to navigate the list. For example, the following code fragment prints all elements of a positional list named data .

```
cursor = data.first()
while cursor is not None:
    print(cursor.element()) # print the element stored at the position
    cursor = data.after(cursor) # advance to the next position
                              (if any)
```

This code relies on the stated convention that the None object is returned when after is called upon the last position. That return value is clearly distinguishable from any legitimate position. The positional list ADT similarly indicates that the None value is returned when the before method is invoked at the front of the list, or when first or last methods are called upon an empty list. Therefore, the above code fragment works correctly even if the data list is empty.

Because the ADT includes support for Python's iter function, users may rely on the traditional for-loop syntax for such a forward traversal of a list named data .

```
for e in data: print(e)
```

More general navigational and update methods of the positional list ADT are shown in the following example.

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

7.4.2 Doubly Linked List Implementation

In this section, we present a complete implementation of a `PositionalList` class using a doubly linked list that satisfies the following important proposition.

Proposition 7.2: Each method of the positional list ADT runs in worst-case $O(1)$ time when implemented with a doubly linked list. We rely on the `DoublyLinkedBase` class from Section 7.3.1 for our low-level representation; the primary responsibility of our new class is to provide a public interface in accordance with the positional list ADT. We begin our class definition in Code Fragment 7.14 with the definition of the public `Position` class, nested within our `PositionalList` class. `Position` instances will be used to represent the locations of elements within the list. Our various `PositionalList` methods may end up creating redundant `Position` instances that reference the same underlying node (for example, when `first` and `last` are the same). For that reason, our `Position` class defines the `__eq__` and `__ne__` special methods so that a test such as `p == q` evaluates to `True` when two positions refer to the same node.

Validating Positions

Each time a method of the `PositionalList` class accepts a position as a parameter, we want to verify that the position is valid, and if so, to determine the underlying node associated with the position. This functionality is implemented by a non-public method

named `validate`. Internally, a position

`n` maintains a reference to the associated node of the linked list, and also a reference to the list instance that contains the specified node. With the container reference, we can robustly detect when a caller sends a position instance that does not belong to the indicated list.

We are also able to detect a position instance that belongs to the list, but that refers to a node that is no longer part of that list. Recall that the `delete node` of the base class sets the previous and next references of a deleted node to `None`; we can recognize that condition to detect a deprecated node.

Access and Update Methods

The access methods of the `PositionalList` class are given in Code Fragment 7.15 and the update methods are given in Code Fragment 7.16. All of these methods trivially adapt the underlying doubly linked list implementation to support the public interface of the positional list ADT. Those methods rely on the `validate` utility to “unwrap” any position that is sent. They also rely on a `make position` utility to “wrap” nodes as `Position` instances to return to the user, making sure never to return a position referencing a sentinel. For convenience, we have overridden the inherited `insert` between utility method so that ours returns a *position* associated with the newly created node (whereas the inherited version returns the node itself).

7.5 Sorting a Positional List

In Section 5.5.2, we introduced the *insertion-sort* algorithm, in the context of an array-based sequence. In this section, we develop an implementation that operates on a PositionalList, relying on the same high-level algorithm in which each element is placed relative to a growing collection of previously sorted elements.

We maintain a variable named *marker* that represents the rightmost position of the currently sorted portion of a list. During each pass, we consider the position just past the marker as the pivot and consider where the pivot's element belongs relative to the sorted portion; we use another variable, named *walk*, to move leftward from the marker, as long as there remains a preceding

element with value larger than the pivot's. A typical configuration of these variables is diagrammed in Figure 7.15. A Python implementation of this strategy is given in Code 7.17.

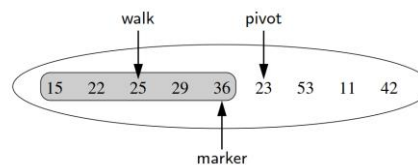


Figure 7.15: Overview of one step of our insertion-sort algorithm. The shaded elements, those up to and including marker, have already been sorted. In this step, the pivot's element should be relocated immediately before the walk position.

7.6 Case Study: Maintaining Access Frequencies

The positional list ADT is useful in a number of settings. For example, a program that simulates a game of cards could model each person's hand as a positional list (Exercise P-7.47). Since most people keep cards of the same suit together, inserting and removing cards from a person's hand could be implemented using the methods of the positional list ADT, with the positions being determined by a natural order of the suits. Likewise, a simple text editor embeds the notion of positional insertion and deletion, since such editors typically perform all updates relative to a *cursor*, which represents the current position in the list of characters of text being edited.

In this section, we consider 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. Examples of such scenarios include a Web 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. We model this with a new *favorites list ADT* that supports the `len` and `is empty` methods as well as the following:

`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.

-

7.6.1 Using a Sorted List

Our first approach for managing a list of favorites is to store elements in a linked list, keeping them in nonincreasing order of access counts. We access or remove an element by searching the

list from the most frequently accessed to the least frequently accessed. Reporting the top k most accessed elements is easy, as they are the first k entries of the list.

To maintain the invariant that elements are stored in nonincreasing order of access counts, we must consider how a single access operation may affect the order. The accessed element's count increases by one, and so it may become larger than one or more of its preceding neighbors in the list, thereby violating the invariant. Fortunately, we can reestablish the sorted invariant using a technique similar to a single pass of the insertion-sort algorithm, introduced in the previous section. We can perform a backward traversal of the list, starting at the position of the element whose access count has increased, until we locate a valid position after which the element can be relocated.

Using the Composition Pattern

We wish to implement a favorites list by making use of a `PositionalList` for storage. If elements of the positional list were simply elements of the favorites list, we would be challenged to maintain access counts and to keep the proper count with the associated element as the contents of the list are reordered. We use a general object-oriented design pattern, the *composition pattern*, in which we define a single object that is composed of two or more other objects. Specifically, we define a nonpublic nested class, `Item`, that stores the element and its access count as a single instance. We then maintain our favorites list as a `PositionalList` of `Item` instances, so that the access count for a user's element is embedded alongside it in our representation. (An `Item` is never exposed to a user of a `FavoritesList`.)

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

The previous implementation of a favorites list performs the `access(e)` method in time proportional to the index of `e` in the favorites list. That is, if `e` is the k th most popular element in the favorites list, then accessing it takes $O(k)$ time. In many real-life access sequences (e.g., Web pages visited by a user), once an element is accessed it is more likely to be accessed again in the near future. Such scenarios are said to possess *locality of reference*.

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*. To apply this heuristic, each time we access an element we move it all the way to the front of the list. Our hope, of course, is that this element will be accessed again in the near future. Consider, for example, a scenario in which we have n elements and the following series of n^2 accesses:

element 1 is accessed n times

element 2 is accessed n times

...

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

...

each access to element n runs in $O(n)$ time.

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

which is $O(n^3)$.

$\frac{n(n+1)}{2}$,

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
- ...
- 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 move-to-front implementation has faster access times for this scenario. Still, the move-to-front approach is just a heuristic, for there are access sequences where using the move-to-front approach is slower than simply keeping the favorites list ordered by access counts.

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 $\Theta(\log n)$. (Recall the big- Ω 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 .

Implementing the Move-to-Front Heuristic in Python

We give an implementation of a favorites list using the move-to-front heuristic in Code Fragment 7.20. The new `FavoritesListMTF` class inherits most of its functionality from the original `FavoritesList` as a base class.

By our original design, the access method of the original class relies on a non-public utility named `move up` to enact the potential shifting of an element forward in the list, after its access count had been incremented. Therefore, we implement the move-to-front heuristic by simply overriding the `move up` method so that each accessed element is moved directly to the front of the list (if not already there). This action is easily implemented by means of the positional list ADT.

The more complex portion of our `FavoritesListMTF` class is the new definition for the `top` method. We rely on the first of the approaches outlined above, inserting copies of the items into a temporary list and then repeatedly finding, reporting, and removing an element that has the largest access count of those remaining.

7.7 Link-Based vs. Array-Based Sequences

We close this chapter by reflecting on the relative pros and cons of array-based and link-based data structures that have been introduced thus far. 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

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 `LinkedList` (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.

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

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.

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).

As an example application, consider a text editor that maintains a document as a sequence of characters. Although users often add characters to the end of the document, it is also possible to use the cursor to insert or delete one or more characters at an arbitrary position within the document. If the character sequence were stored in an array-based sequence (such as a Python list), each such edit operation may require linearly many characters to be shifted, leading to $O(n)$ performance for each edit operation. With a linked-list representation, an arbitrary edit operation (insertion or deletion of a character at the cursor) can be performed in $O(1)$ worst-case time, assuming we are given a position that represents the location of the cursor.

END

六、实验体会

七、参考文献

- [1] Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser, Data Structures and Algorithms in Python
- [2] 数据结构与算法分析：C 语言描述（原书第二版），（美）维斯著；冯舜玺译. 北京：机械工业出版社
- [3] 算法导论（原书第三版），（美）科尔曼（Cormen, T.H.）等；殷建平等译. 北京：机械工业出版社