Data structure and algorithm in Python

Linked Lists

Xiaoping Zhang

School of Mathematics and Statistics, Wuhan University

Table of contents

Python's list class is highly optimized, and often a great choice for storage. With that said, there are some notable disadvantages:

1. The length of a dynamic array might be longer than the actual number of elements that it stores.
2. Amortized bounds for operations may be unacceptable in real-time systems.
3. Insertions and deletions at interior positions of an array are expensive.

In this lecture, 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.

Singly Linked Lists

Singly Linked Lists

Definition

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

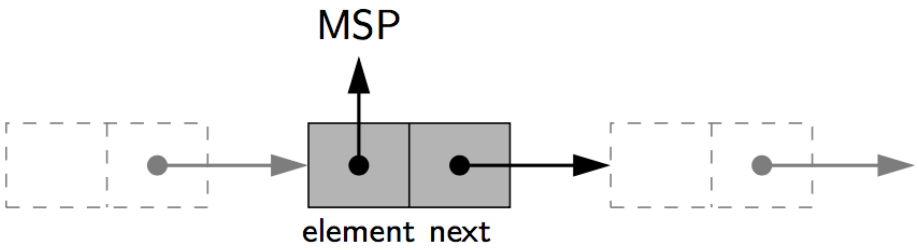


Figure 7.1: Example of a node instance that forms part of a singly linked list. The node's element member references an arbitrary object that is an element of the sequence (the airport code MSP, in this example), while the next member references the subsequent node of the linked list (or None if there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

Singly Linked Lists

Definition : head and tail

The first and last node of a linked list are known as the head and tail of the list, respectively.

Definition : traverse

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.

Singly Linked Lists

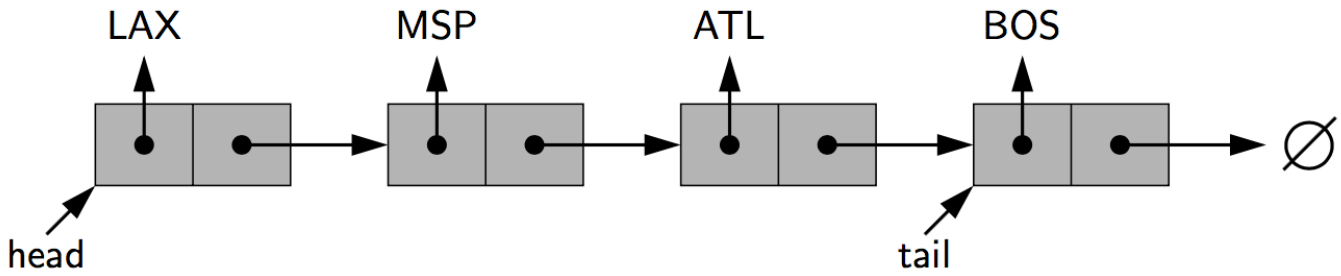


Figure 7.2: Example of a singly linked list whose elements are strings indicating airport codes. The list instance maintains a member named head that identifies the first node of the list, and in some applications another member named tail that identifies the last node of the list. The **None** object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

Singly Linked Lists

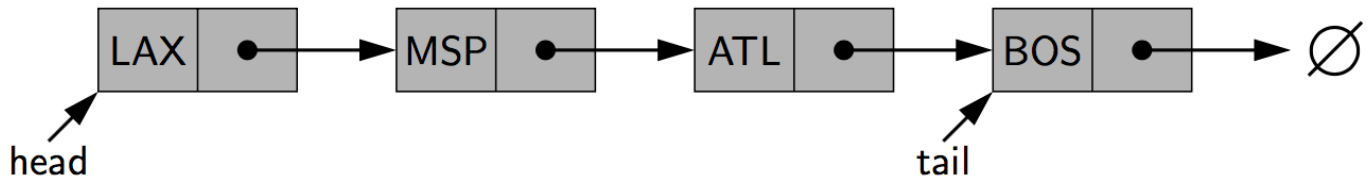


Figure 7.3: A compact illustration of a singly linked list, with elements embedded in the nodes (rather than more accurately drawn as references to external objects).

Singly Linked Lists

Inserting an Element at the Head of a Singly Linked List

Inserting an Element at the Head of a Singly Linked List

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.

Inserting an Element at the Head of a Singly Linked List

How to insert an element at the head of the list?

Inserting an Element at the Head of a Singly Linked List

How to insert an element at the head of the list?

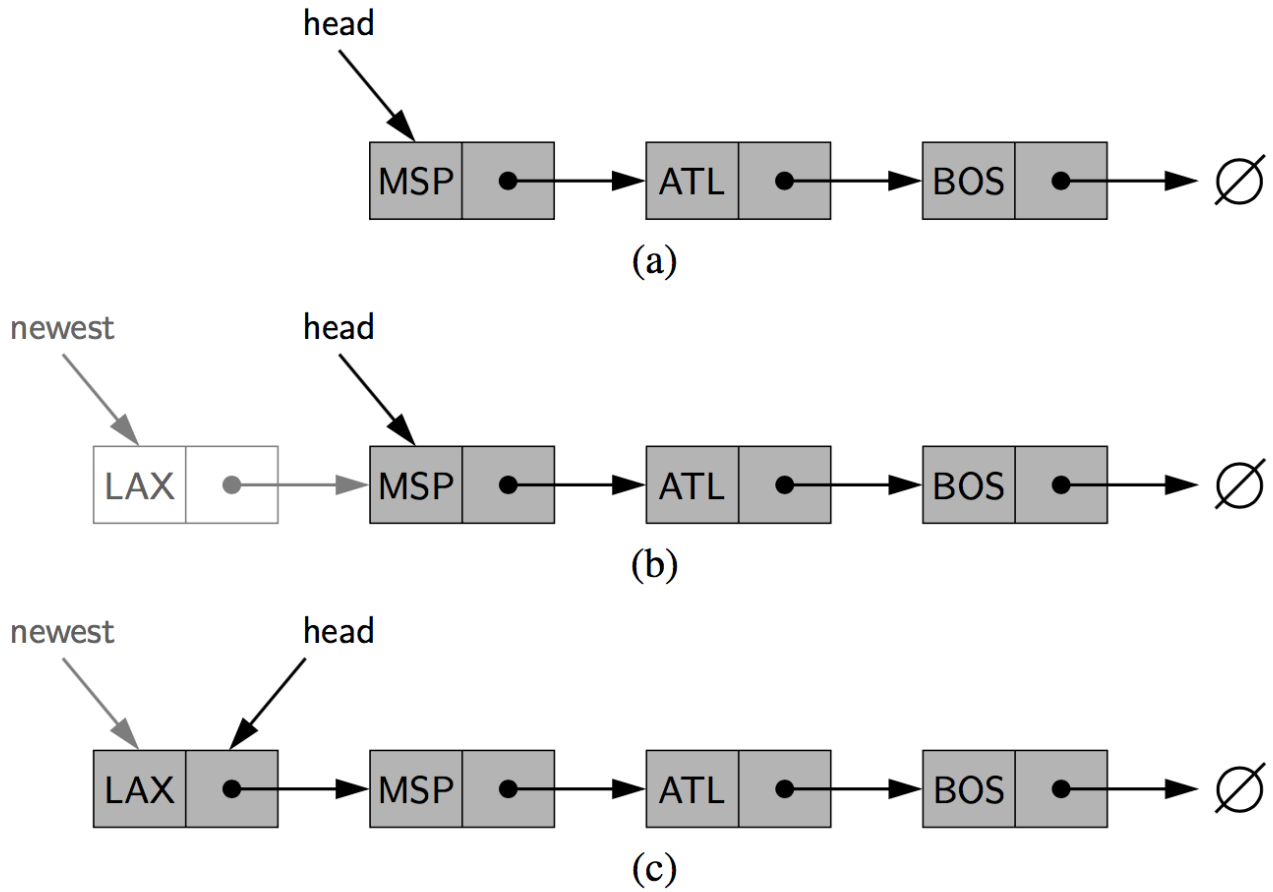


Figure 7.4: Insertion of an element at the head of a singly linked list: (a) before the insertion; (b) after creation of a new node; (c) after reassignment of the head reference.

Inserting an Element at the Head of a Singly Linked List

1. create a new node,

   set its element to the new element,

   set its next link to the current head;

2. set the list's head to point to the new node.

Inserting an Element at the Head of a Singly Linked List

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.

Singly Linked Lists

Inserting an Element at the tail of a Singly Linked List

Inserting an Element at the tail of a Singly Linked List

We can also easily insert an element at the tail of the list, provided we keep a reference to the tail node.

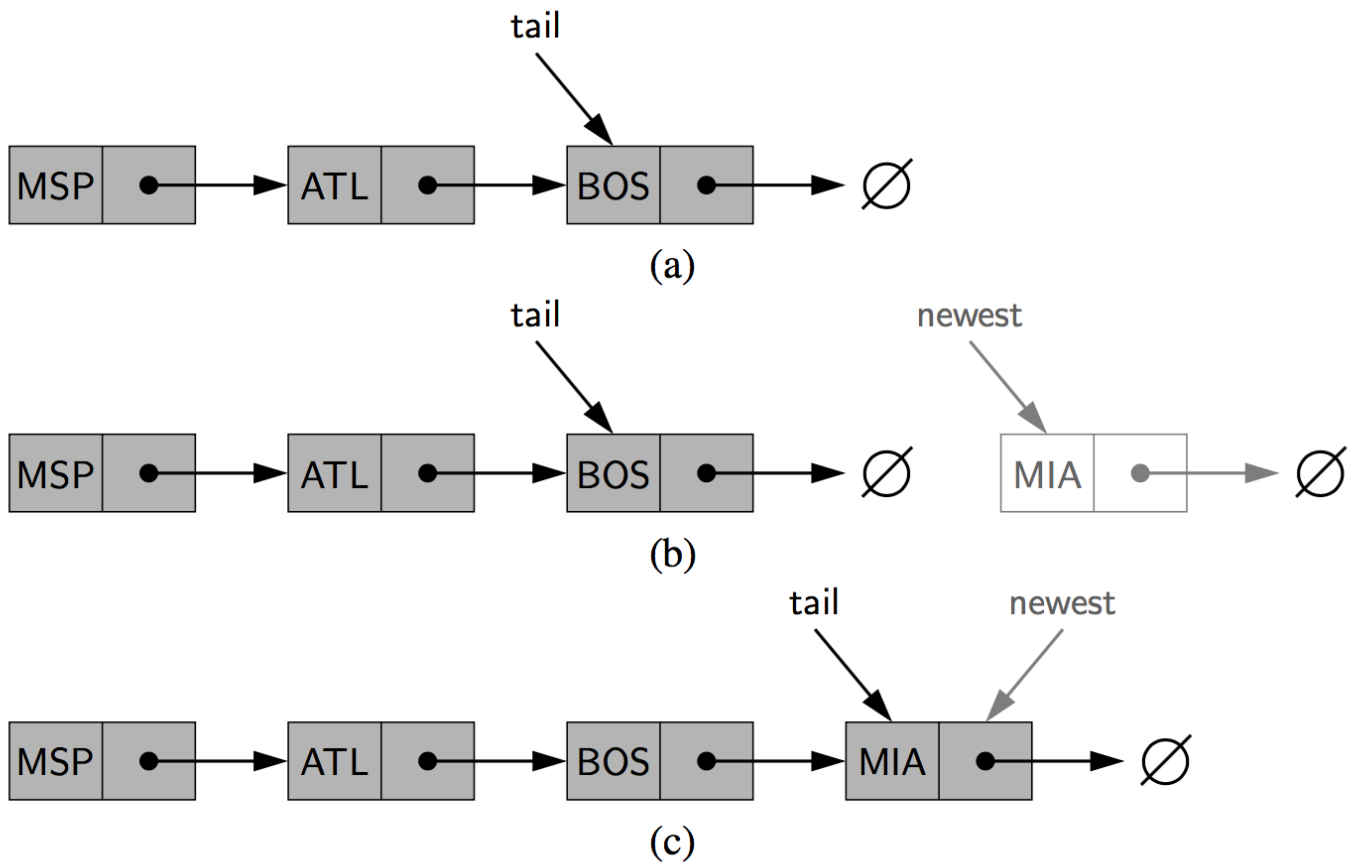


Figure 7.5: Insertion at the tail of a singly linked list: (a) before the insertion; (b) after creation of a new node; (c) after reassignment of the tail reference. Note that we must set the next link of the tail in (b) before we assign the tail variable to point to the new node in (c).

Inserting an Element at the tail of a Singly Linked List

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.

Singly Linked Lists

Removing an Element at the head of a Singly Linked List

Removing an Element at the head of a Singly Linked List

Removing an element from the head of a singly linked list is essentially the reverse operation of inserting a new element at the head.

Removing an Element at the head of a Singly Linked List

Removing an element from the head of a singly linked list is essentially the reverse operation of inserting a new element at the head.

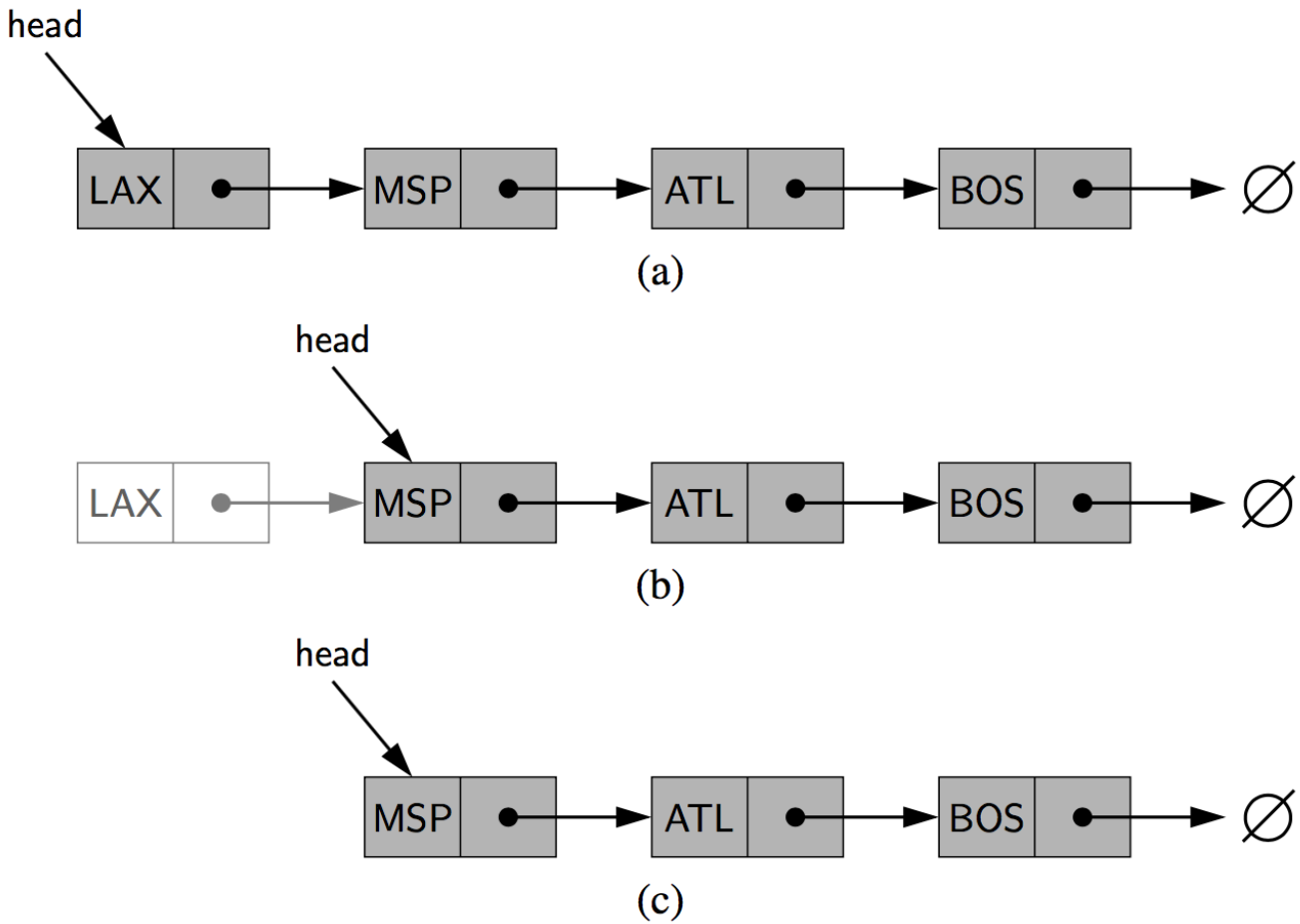


Figure 7.6: Removal of an element at the head of a singly linked list: (a) before the removal; (b) after "linking out" the old head; (c) final configuration.

Removing an Element at the head 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.

Removing an Element at the head 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.

Singly Linked Lists

Implementing a Stack with a Singly Linked List

Implementing a Stack with a Singly Linked List

To implement a stack with singly linked list, we need to decide whether to model the top of stack at the head or at the tail of the list.

Implementing a Stack with a Singly Linked List

To implement a stack with singly linked list, we need to decide whether to model the top of stack at the head or at the tail of the list.

We can orient the top of the stack at the head because we can efficiently insert and delete elements in constant time only at the head as well as all stack operations affect the top.

Implementing a Stack with a Singly Linked List

```
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
```

Implementing a Stack with a Singly Linked List

```
def __len__(self):
  return self._size

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

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

def top(self):
  if self.is_empty():
    raise Empty('Stack is empty')
  return self._head._element
```

Implementing a Stack with a Singly Linked List

```
def pop(self):
  if self.is_empty():
    raise Empty('Stack is empty')
  answer = self._head._element
  self._head = self._head._next
  self._size -= 1
  return answer
```

Implementing a Stack with a Singly Linked List

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.

Singly Linked Lists

Implementing a Queue with a Singly Linked List

Implementing a Queue with a Singly Linked List

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.

Implementing a Queue with a Singly Linked List

```
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
```

Implementing a Queue with a Singly Linked List

```
def __len__(self):
  return self._size

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

def first(self):
  if self.is_empty():
    raise Empty('Queue is empty')
  return self._head._element
```

Implementing a Queue with a Singly Linked List

```
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
```

Implementing a Queue with a Singly Linked List

```
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
```

Implementing a Queue with a Singly Linked List

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

Circularly Linked Lists

Circularly Linked Lists

Definition : Circularly Linked Lists

A circularly linked list is a linked list that its tail use its next reference to point back to the head of the list.

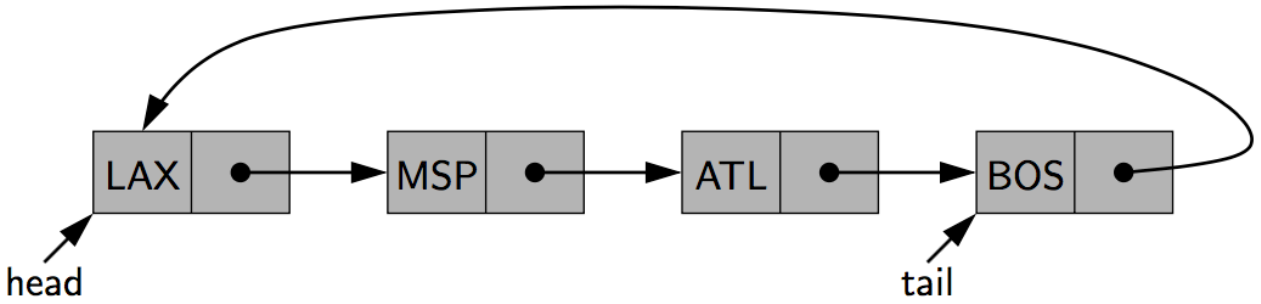


Figure 7.7: Example of a singly linked list with circular structure.

Circularly Linked Lists

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.

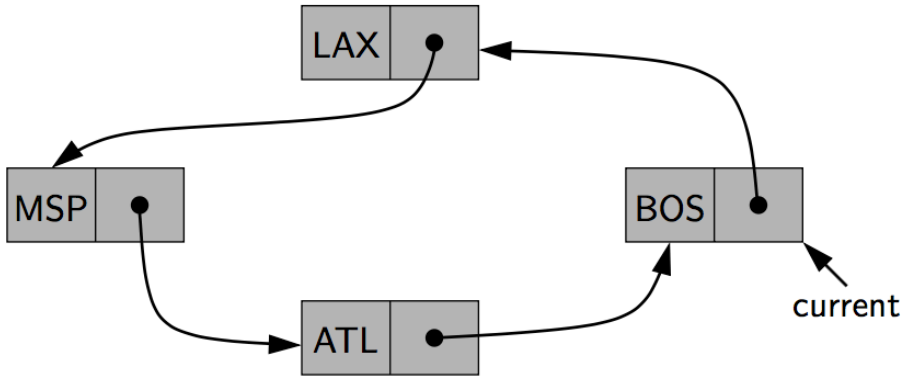


Figure 7.8: Example of a circular linked list, with current denoting a reference to a select node.

Circularly Linked Lists

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.

Circularly Linked Lists

Implementing a Queue with a Circularly Linked List

Implementing a Queue with a Circularly Linked List

```
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
```

Implementing a Queue with a Circularly Linked List

```
def __len__(self):
  return self._size

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

def first(self):
  if self.is_empty():
    raise Empty('Queue is empty')
  head = self._tail._next
  return head._element
```

Implementing a Queue with a Circularly Linked List

```
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
```

Implementing a Queue with a Circularly Linked List

```
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
```

Doubly Linked Lists

Doubly Linked Lists

Definition : Doubly Linked Lists

A doubly linked list is a linked list in which each node keeps an explicit reference to the node before it and a reference to the node after it.

Doubly Linked Lists

Definition : Doubly Linked Lists

A doubly linked list is a linked list in which each node keeps an explicit reference to the node before it and a reference to the node after it.

These lists allow a greater variety of $O(1)$-time update operations, including insertions and deletions at arbitrary positions within the list.

Doubly Linked Lists

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.

Doubly Linked Lists

Header and Trailer Sentinels

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.

Header and Trailer Sentinels

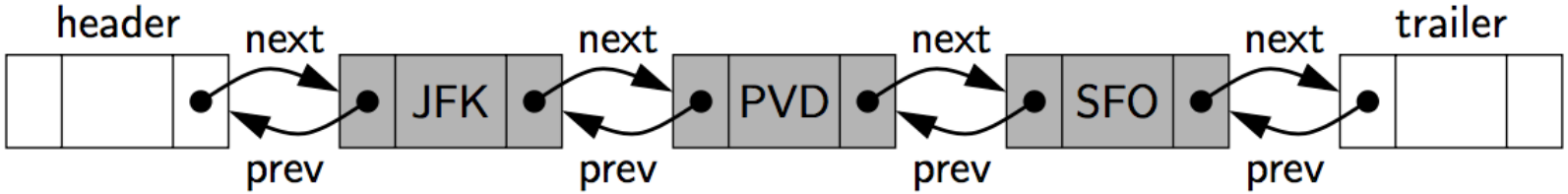


Figure 7.10: A doubly linked list representing the sequence { JFK, PVD, SFO }, using sentinels header and trailer to demarcate the ends of the list.

Header and Trailer Sentinels

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 set None;
- 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.

Doubly Linked Lists

Advantage of Using Sentinels

Advantage of Using Sentinels

- Although we could implement a doubly linked list without sentinel nodes, 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.

- 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.

Doubly Linked Lists

Inserting and Deleting with a Doubly Linked List

Inserting and Deleting with a Doubly Linked List

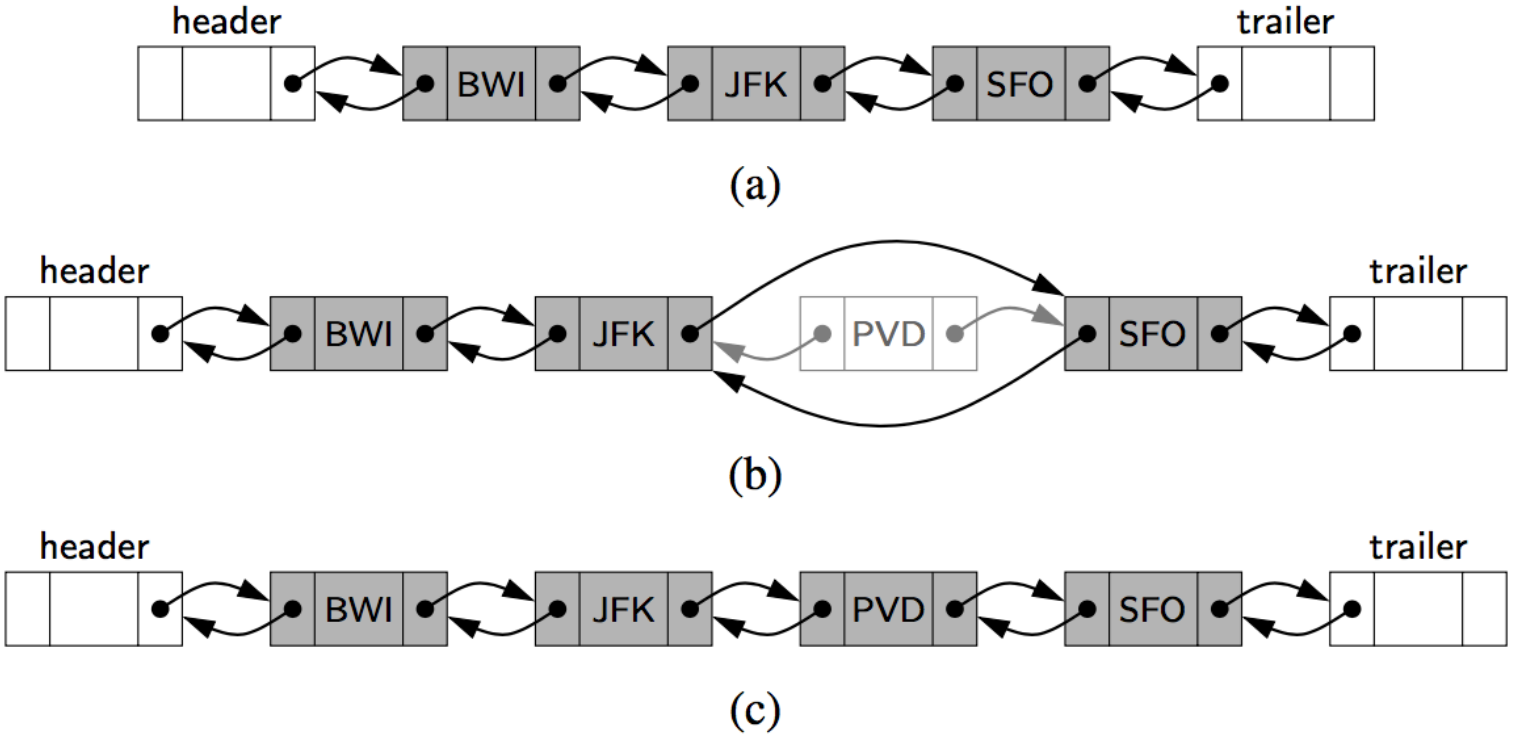


Figure 7.11: Adding an element to a doubly linked list with header and trailer sentinels: (a) before the operation; (b) after creating the new node; (c) after linking the neighbors to the new node.

Inserting and Deleting with a Doubly Linked List

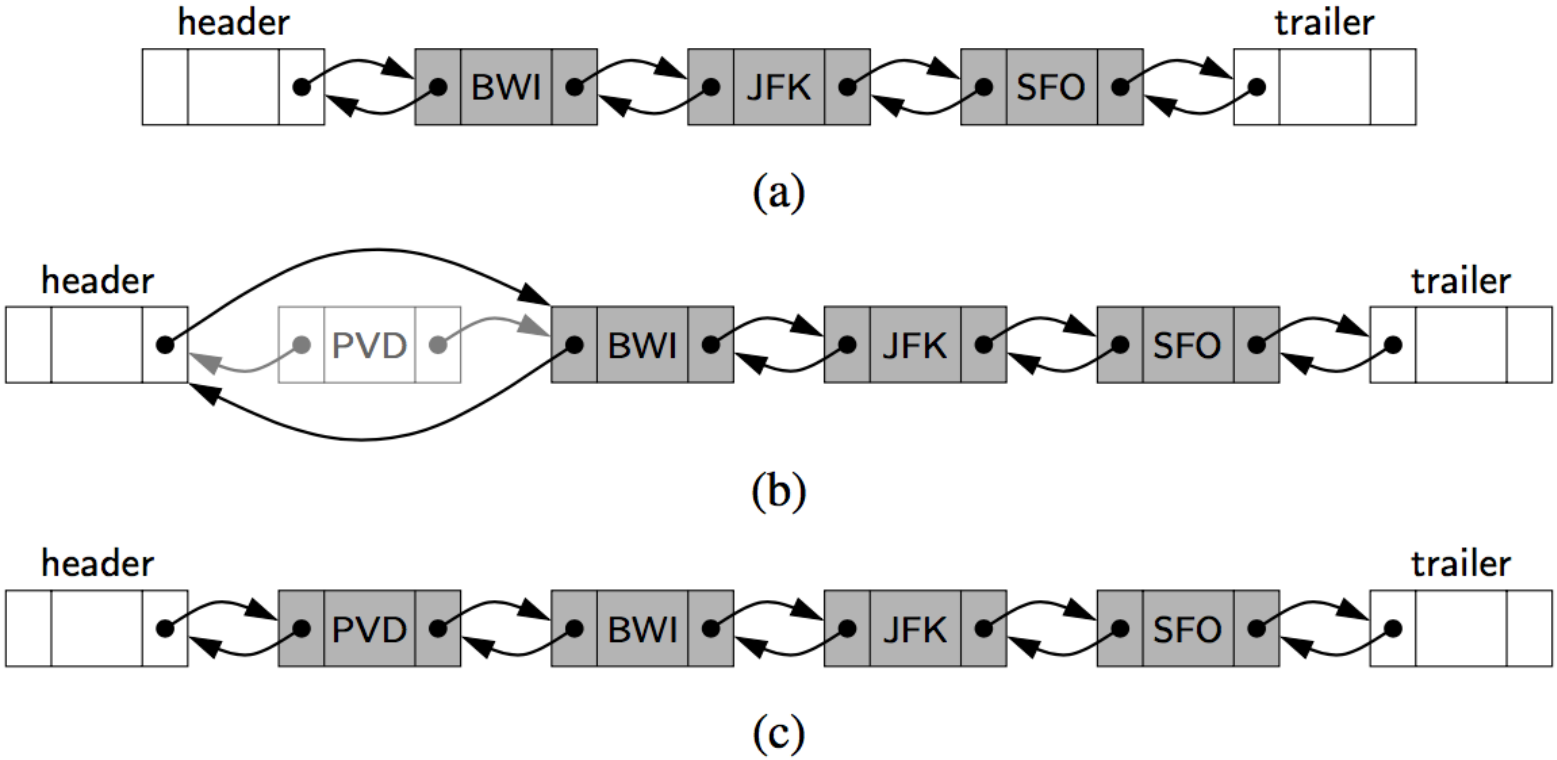


Figure 7.12: Adding an element to the front of a sequence represented by a doubly linked list with header and trailer sentinels: (a) before the operation; (b) after creating the new node; (c) after linking the neighbors to the new node.

Inserting and Deleting with a Doubly Linked List

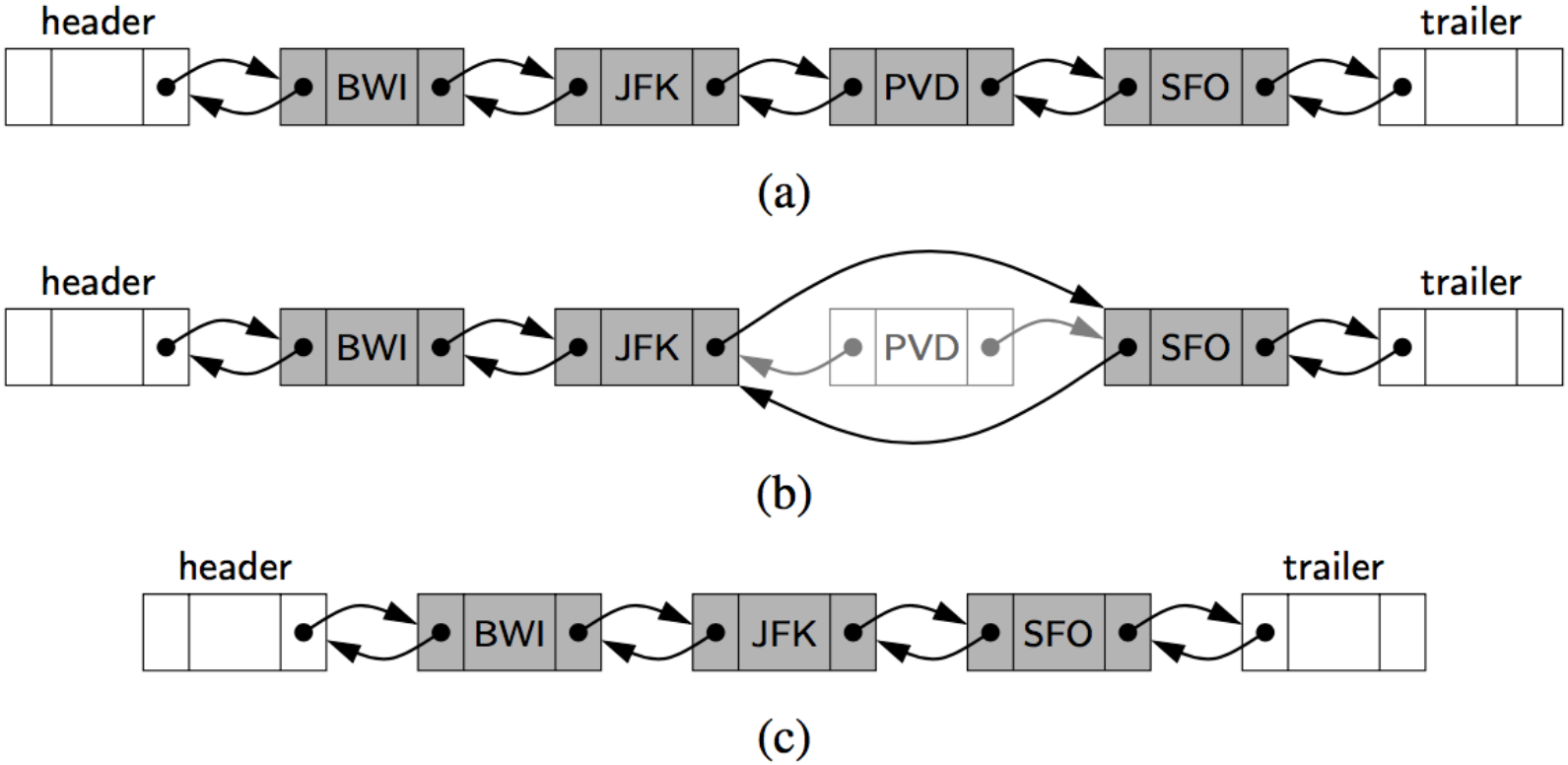


Figure 7.13: Removing the element PVD from a doubly linked list: (a) before the removal; (b) after linking out the old node; (c) after the removal (and garbage collection).

Doubly Linked Lists

Basic Implementation of a Doubly Linked List

Basic Implementation of a Doubly Linked List

```
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
```

Basic Implementation of a Doubly Linked List

```
def __len__(self):
  return self._size

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

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

Basic Implementation of a Doubly Linked List

```
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
```

Doubly Linked Lists

Implementing a Deque with a Doubly Linked List

For double-ended queue (deque),

- 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.

Implementing a Deque with a Doubly Linked List

```
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

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

Implementing a Deque with a Doubly Linked List

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

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
  )
```

The Positional List ADT

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.

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.

The Positional List ADT

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.

The Positional List ADT

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?

The Positional List ADT

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.

The Positional List ADT

When working with array-based sequences, 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.

The Positional List ADT

When working with array-based sequences, 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.

The Positional List ADT

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.

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.

The Positional List ADT

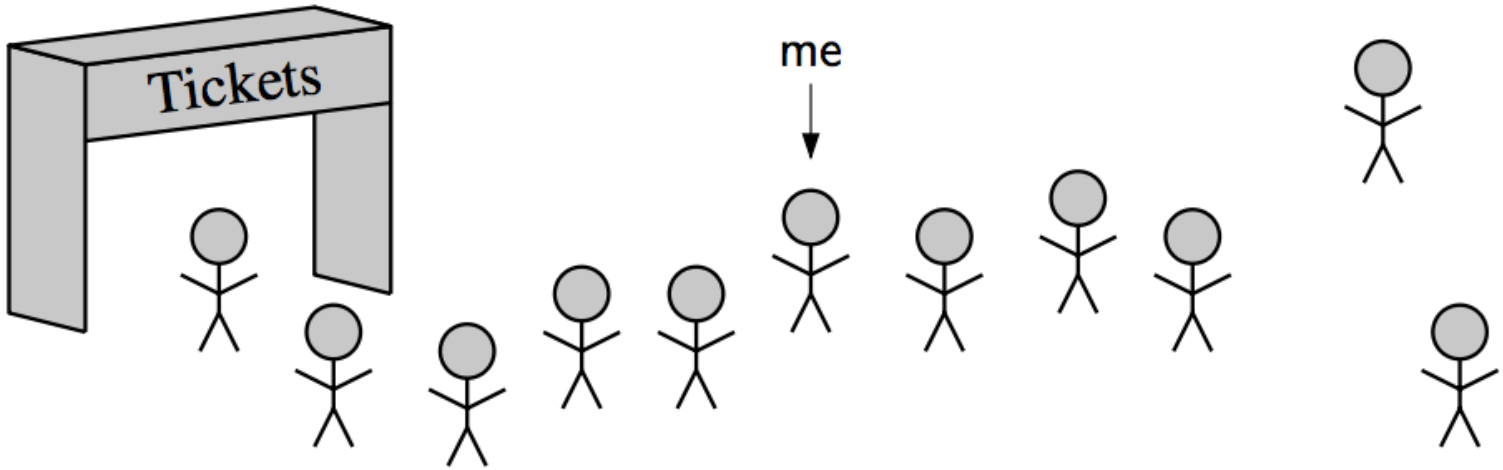


Figure 7.14: We wish to be able to identify the position of an element in a sequence without the use of an integer index.

The Positional List ADT

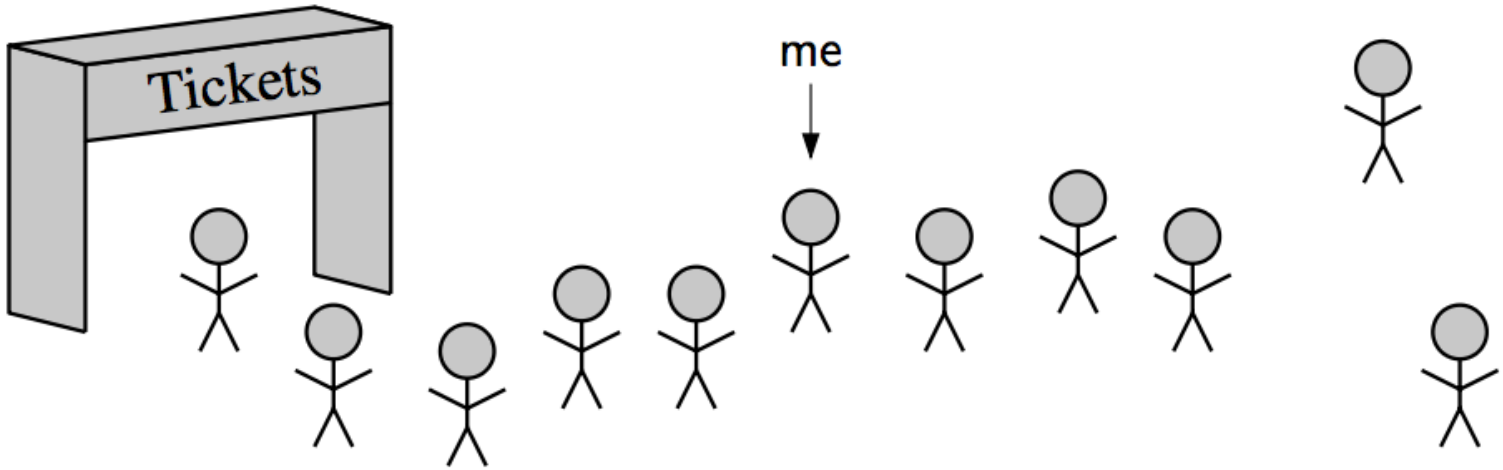


Figure 7.14: We wish to be able to identify the position of an element in a sequence without the use of an integer index.

We 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.

The Positional List ADT

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, allow operations such as "delete the character at the cursor" or "insert a new character just after the cursor".

The Positional List ADT

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.

The Positional List ADT

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 references serves as the mechanism for describing a position.

The Positional List ADT

The Positional List Abstract Data Type

The Positional List Abstract Data Type

To provide for a general abstraction of a sequence of elements with the ability to indetify the location of an element, we define a positional list ADT as well as a simple position ADT 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 exlicit command is issued to delete it.

The Positional List Abstract Data Type

A position instance is a simple object, supporting only the following method:

- `p.element()`: Return the element stored at position p.

The Positional List Abstract Data Type

A position instance is a simple object, supporting only the following method:

- `p.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.

The Positional List Abstract Data Type

In describing the behaviors of a positional list, we being by presenting the accssor methods supported by a list L:

- `L.first()`: Return the position of the first element of L, or None if L is empty.
- `L.last()`: Return the position of the last element of L, or None if L is empty.
- `L.before(p)`: Return the position of L immediately before position p, or None if p is the first position.
- `L.after(p)`: Return the position of L immediately after position p, or None if p is the last position.
- `L.is_empty()`: Return true if list L does not contain any elements.
- `len(L)`: Return the number of elements in the list.
- `iter(L)`: Return a forward iterator for the elements of the list.

The Positional List Abstract Data Type

The positional list ADT also includes the following update methods:

- `L.add_firt(e)`: Insert a new element e at the front of L, returning the position of the new element.
- `L.add_last(e)`: Insert a new element e at the back of L, returning the position of the new element.
- `L.add_before(p, e)`: Insert a new element e just before position p in L, returning the position of the new element.
- `L.add_after(p, e)`: Insert a new element e just after position p in L, returning the position of the new element.
- `L.replace(p, e)`: Replace the element at position p with element e, returning the element formerly at position p.
- `L.delete(p)`: Remove and return the element at position p in L, invalidating the position.

The Positional List Abstract Data Type

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	$9_s, 7_p, 3_r$

The Positional List ADT

Doubly Linked List Implementation

Doubly Linked List Implementation

We now present a complete implementation of a PositionalList class using a doubly linked list that satisfies the following important proposition.

Property 4.1

Each method of the positional list ADT runs in worst-case $O(1)$ time when implemented with a doubly linked list.

Doubly Linked List Implementation

```
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)
```

The Positional List ADT

```
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

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

The Positional List ADT

```
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)
```

The Positional List ADT

```
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)
```

The Positional List ADT

```
  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)
```

The Positional List ADT

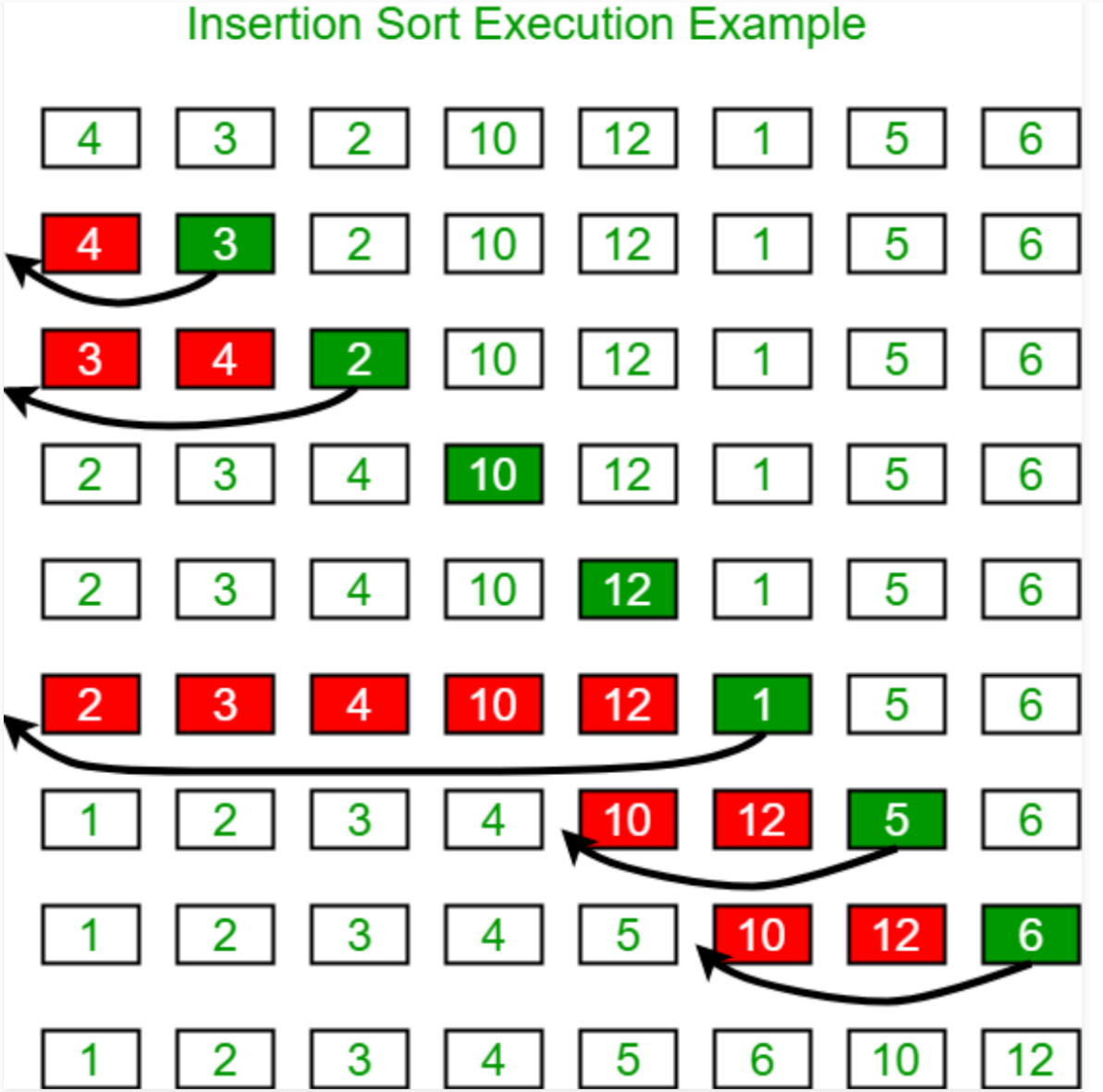
```
def add_before(self, p, e):
  original = self._validate(p)
  return self._insert_between(e, original.
  _prev, original)

def add_after(self, p, e):
  original = self._validate(p)
  return self._insert_between(e, original,
  original._next)

def delete(self, p):
  original = self._validate(p)
  return self._delete_node(original)
```

The Positional List ADT

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

Sorting a Positional List

Sorting a Positional List



Sorting a Positional List

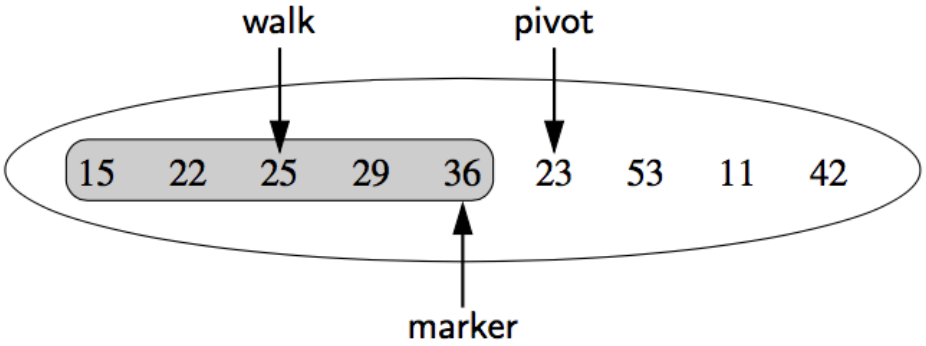


Figure 7.15: Overview of one step of our insertion-sort algorithm. The shaded elements, those up to and including marker, have already been sorted. In this step, the pivot's element should be relocated immediately before the walk position.

Sorting a Positional List

```
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)
```

Favorite List

Favorite List

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.

Example

- A Web browser keeps track of a user's most accessed URLs,
- A music collection maintains a list of the most frequently played songs for a user.
- .....

Favorite List

We model this with a new favorites list ADT that support 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.

Favorite List

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

Favorite List

We implement a favorites list by making use of a PositionalList for storage.

We define a nonpublic class `_Item` to store the element and its access count as a single instance, and 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.

Favorite List

```
from positional_list import PositionalList
class FavoritesList:
  class _Item:
    __slots__ = '_value', '_count'
    def __init__(self, e):
      self._value = e
      self._count = 0
```

Favorite List

```
def __init__(self):
  self._data = PositionalList()

def __len__(self):
  return len(self._data)

def is_empty(self):
  return len(self._data) == 0
```

Favorite List

```
def _find_position(self, e):
  walk = self._data.first()
  while walk is not None and walk.element().
  _value != e:
    walk = self._data.after(walk)
  return walk
```

Favorite List

```
def _move_up(self, p):
  if p != self._data.first():
    cnt = p.element()._count
    walk = self._data.before(p)
    if cnt > walk.element()._count:
      while (walk != self._data.first() and
             cnt > self._data.before(walk).
             element()._count):
        walk = self._data.before(walk)
      self._data.add_before(walk, self._data.
      delete(p))
```

Favorite List

```
def access(self, e):
  p = self._find_position(e)
  if p is None:
    p = self._data.add_last(self._Item(e))
  p.element()._count += 1
  self._move_up(p)

def remove(self, e):
  p = self._find_position(e)
  if p is not None:
    self._data.delete(p)
```

Favorite List

```
def top(self, k):
  if not 1 <= k <= len(self):
    raise ValueError('Illegal value for k')
  walk = self._data.first()
  for j in range(k):
    item = walk.element()
    yield item._value
    walk = self._data.after(walk)
```

Favorite List

```
  def __repr__(self):
    return ', '.join('({0}:{1})'.format(i._value
    , i._count) for i in self._data)

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