Data Structures and Algorithms in Python

Michael T. Goodrich
Department of Computer Science
University of California, Irvine

Roberto Tamassia
Department of Computer Science
Brown University

Michael H. Goldwasser
Department of Mathematics and Computer Science
Saint Louis University

Instructor's Solutions Manual

WILEY

Chapter 7 Linked Lists

Hints and Solutions

Reinforcement

R-7.1) Hint It is okay to have an algorithm running in linear time.

R-7.1) Solution While the precise solution depends upon the linked list representation, the following is a typical approach.

```
def penultimate_node(the_list):
  if len(the_list) < 2:
     raise ValueError('list must have 2 or more entries')
  walk = the_list._head
  while walk._next._next is not None:
    walk = walk._next
  return walk
```

R-7.2) Hint This concatenation operation need not search all of L and M.

R-7.2) Solution Simply use a temporary node to walk to the end of list L. Then, make the last element of L point to the first element of M as its "next" node.

```
def concatenate(L,M):
  walk = L._head
  while walk._next is not None:
    walk = walk._next
  walk._next = M._head
  M._head = None
```

R-7.3) Hint Consider passing a node as a parameter.

R-7.3) Solution We can call the following function on the head node of a list.

```
def count(node):
  if node is None:
    return 0
  else:
    return 1 + count(node._next)
```

R-7.4) Hint Performing the swap for a singly linked list will take longer than for a doubly linked list.

R-7.4) Solution Implementing a precise solution takes great care, especially when x and y neighbor each other. However, the issue regarding efficiency is that for swapping x and y in a singly-linked, we must locate the nodes immediately preceding x and y, and we have no quick way to do so.

R-7.5) Hint You need to keep track of where you start or your method will have an infinite loop.

R-7.5) Solution As an example solution, here is an implementation of the __len__ method for our CircularQueue class that does not presume use of a precalculated _size member.

```
def __len__(self):
  if self._tail is None:
    return 0
  walk = self._tail._next
  count = 1
  while walk != self._tail:
    count += 1
    walk = walk._next
  return count
```

R-7.6) Hint Your only need to go around one of the lists once.

R-7.6) Solution Starting at x._next, walk forward until either encountering x or y. If y is first reached, then they are in the same list; if x is first reach than y belongs to some other list.

R-7.7) Hint You must adjust links so that the first node is moved to the end of the list.

R-7.7) Solution

```
def rotate(self):
  if not self.is_empty( ):
    self._tail._next = self._head
    self._head = self._head._next
    self._tail = self._tail._next
    self._tail._next = None
```

R-7.8) Hint Consider a combined search from both ends. Also, recall that a link hop is an assignment of the form "p = p._next" or "p = p._prev".

R-7.8) Solution The following solution, implemented as a method of the _DoublyLinkedBase class, runs in $O(n)$ time.

```
def middle(self):
  if self._size == 0:
    raise ValueError('list must be nonempty')
  middle = self._header._next
  partner = self._trailer._prev
  while middle != partner and middle._next != partner:
    middle = middle._next
    partner = partner._prev
  return middle
```

R-7.9) Hint Splice the end of L into the beginning of M.

R-7.9) Solution Use two temporary Node elements, temp1 and temp2. Initialize temp1 to be the trailer node of L and temp2 to be the header node of M. Make the element of temp1 have its next field point to temp2 and set the element of temp2 to have its prev field point to temp1. Set L' to be L and then set the trailer node of L' to be the trailer node of M.

R-7.10) Hint Is there a scenario in which these substitions fail?

R-7.10) Solution The substitutions will not work when the list is empty.

R-7.11) Hint Keep track of the maximum thus far while walking the list

R-7.11) Solution

```
def max(the_list):
  if len(the_list) == 0:
    raise ValueError('list must be nonempty')
  best = walk = the_list.first( )
  while walk is not None:
    if walk.element( ) > best.element( ):
      best = walk
  return best.element( )
```

R-7.12) Hint Within a method of the class, you may access nonpublic members

R-7.12) Solution

```
def max(self):
  if self._size == 0:
    raise ValueError('list must be nonempty')
  best = walk = self._header._next
  while walk is not self._trailer:
    if walk._element > best._element:
      best = walk
  return best._element
```

R-7.13) Hint Start looking at the beginning of the list.

R-7.13) Solution

```
def find(self, e):
  walk = self._header._next
  while walk is not self._trailer and walk._element != e:
    walk = walk._next
  if walk is self._trailer:
    return None
  else:
    return self.Position(self, walk)          # return a Position instance
```

R-7.14) Hint Consider parameterizing the method with a node of the list.

R-7.14) Solution

```
def _find_recur(self, e, node):
  if node is self._trailer:
    return None
  elif node._element == e:
    return self.first( )
  else:
    return self._find_recur(e, node._next)       # recurse

def find(self, e):
  return self._find_recur(e, self._header._next)
```

R-7.15) Hint Model your solution on the original implementation with appropriate symmetry.

R-7.15) Solution

```
def __reversed__(self):
  cursor = self.last( )
  while cursor is not None:
    yield cursor.element( )
    cursor = self.before(cursor)
```

R-7.16) Hint Be careful when working with an empty list.

R-7.16) Solution

```
def add_last(self, e):
  if self.is_empty( ):
    return self.add_first(e)
  else:
    return self.add_after(self.last( ), e)

def add_before(self, p, e):
  if p == self.first( ):
    return self.add_first(e)
  else:
    return self.add_after(self.before(p), e)
```

(We don't actually need to use the after(p) method.)

R-7.17) Hint Be careful to repair the list in the neighborhood abandoned by the moved node.

R-7.17) Solution

```
def move_to_front(self, p):
  node = self._validate(p)
  if node != self._header._next:
    # remove node from existing location
    node._prev._next = node._next
    node._next._prev = node._prev
    # make node point to its new neighbors
    node._prev = self._header
    node._next = self._header._next
    # make new neighbors point to the node
    node._prev._next = node
    node._next._prev = node
```

R-7.18) Hint Implement the move-to-front using a pencil and eraser. Better yet, write the six letters on separate pieces of paper and simulate the actions physically.

R-7.18) Solution $\{e,d,b,f,c,a\}$

R-7.19) Hint Consider the two extreme cases of how we could distribute m accesses across n elements.

R-7.19) Solution The minimum is 0, since we could have accessed each element exactly k times. the maximum is $n-1$, since we could have accessed just one element kn times.

R-7.20) Hint The first should be last, both physically and in terms of how long ago it has been accessed.

R-7.21) Hint For this lower bound, assume that when an element is accessed we search for it by traversing the list starting at the front.

R-7.21) Solution For this lower bound, we assume that when an element is accessed, we must search for it by traversing the list starting at the front. We consider n elements, and access them each once in the order $1, 2, \ldots n$, and then access them each again in the original order, repeating n such phases of access. Other than when each element is originally inserted, each of the remaining accesses is to the element at the end of the list, and thus we have $n(n-1)$ accesses which take n steps, thus $\Omega(n^3)$ time overall.

R-7.22) Hint You can either clear the underlying list or start over with a new list.

R-7.23) Hint You will need to adjust instances of the nested _Item class.

R-7.23) Solution

```
def reset_counts(self):
  walk = self._data.first( )
  while walk is not None:
    walk.element( )._count = 0
    walk = self._data.after(walk)
```

Creativity

C-7.24) Hint Admittedly, it is not clear that there is any advantage to the sentinel for this purpose.

C-7.25) Hint You should be able to avoid the conditional within enqueue.

C-7.26) Hint Make sure to leave the head and tail members of both lists with appropriate values.

C-7.26) Solution

```
def concatenate(self, other):
  if len(other) > 0:
    self._tail._next = other._head
    self._tail = other._tail
    self._size += other._size
    other._head = other._tail = None
    other._size = 0
```

C-7.27) Hint View the chain of nodes following the head node as themselves forming another list.

C-7.28) Hint Recur on the first $n-1$ positions.

C-7.28) Solution Let us define a method reverse(L,n), which reverses the first $n \leq \text{len}(L)$ nodes in L, and returns a pointer *end* to the node just after the n^{th} node in L (*end* = **None** if $n = \text{len}(L)$). If $\text{len}(L) \leq 1$, we are done, so let us assume L has at least 2 nodes. If $n = 1$, then we return L.first().next(). Otherwise, we recursively call reverse$(L,n-1)$, and let *end* denote the returned pointer to the n^{th} node in L. We then set *ret* to *end*._next if $n < \text{len}(L)$, and to **None** otherwise. We then insert the node pointed to by *end* at the front of L and we return *ret*. The total running time is $O(n)$.

C-7.29) Hint Consider changing the orientation of links while making a single pass through the list.

C-7.29) Solution Here is an example solution, implemented as a method of a class such as LinkedStack.

```
def reverse(self):
  prev = None
  walk = self._head
  while walk is not None:
    adv = walk._next
    walk._next = prev
    prev = walk
    walk = adv
  self._head = prev
```

Note well that the above implementation works, even for trivial lists.

C-7.30) Hint Think carefully about the orientation of the linked list.

C-7.31) Hint Consider using an abstraction that is a subset of the positional list ADT.

C-7.32) Hint You should replace the first() and last() methods with a method abstracting the cursor.

C-7.33) Hint You will need to carefully switch next and prev pointers and properly manage the sentinels.

C-7.34) Hint Watch out for the special case when p and q are neighbors.

C-7.35) Hint See Section 2.3.4 for discussion of iterators.

C-7.36) Hint Watch out for special cases when the length is one or less.

C-7.37) Hint To get you started, consider if the smallest and largest values add to V. If not, you should be able to eliminate one of the two as unnecessary.

C-7.37) Solution

```
def pairsum(L, V):
  small = L.first( )
  large = L.last( )
  while small != large:
    total = small.element( ) + large.element( )
    if total == V:
      return (small,large)
    elif total < V:
      small = L.after(small)          # previous small was unnecessarily small
    else:    # total > V
      large = L.before(large)         # previous large was unnecessarily large
  return None    # search failed
```

C-7.38) Hint It would be helpful to implement a swap subroutine.
C-7.39) Hint Carefully map the public methods of the queue interface to the concrete behaviors of the PositionalList class.
C-7.40) Hint Note well that there may be fewer than n elements included in the most recent n accesses, due to duplication.
C-7.40) Solution We will maintain a global count of the number of accesses in the sequence, and then with each element in the list, we will keep track of the time stamp for its most recent access. After each operation is performed, we examine the last item in the list and if its most recent access is no longer within the most recent n operations, that element is deleted. (Note that only one element can be purged after a single operation, as it can be the only one who's most recent access was precisely $n+1$ operations ago.)
C-7.41) Hint Be sure to handle the case where every pair (x,y) in A and every pair (y,z) in B have the same y value.
C-7.42) Hint You should keep track of the number of game entries explicitly.
C-7.43) Hint Convert the two parts to two separate lists as sublists.

Projects

P-7.44) Hint Use a position instance variable to keep track of the cursor location.
P-7.45) Hint There is a trade-off between insertion and searching depending on whether the entries in L are sorted.
P-7.46) Hint It is okay to be inefficient in this case.
P-7.47) Hint Keep all cards in a single list, and use four positions to demark the beginning of the respective suits.