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 6 Stacks, Queues, and Deques

Hints and Solutions

Reinforcement

R-6.1) Hint Use a paper and pencil with eraser to simulate the stack.

R-6.1) Solution 3, 8, 2, 1, 6, 7, 4, 9

R-6.2) Hint If a stack is empty when pop is called, its size does not change.

R-6.2) Solution The size of the stack is $25 - 10 + 3 = 18$.

R-6.3) Hint Transfer items one at a time.

R-6.3) Solution

```
def transfer(S, T):
  while not S.is_empty( ):
    T.push(S.pop( ))
```

R-6.4) Hint First check if the stack is already empty.

R-6.4) Solution If the stack is empty, then return (the stack is empty). Otherwise, pop the top element from the stack and recur.

R-6.5) Hint Use two loops.

R-6.5) Solution

```
def reverse(sequence):
  """A nonrecursive method for reversing a Python list."""
  S = ArrayStack( )
  for j in range(len(sequence)):
    S.push(sequence[j])
  for j in range(len(sequence)):
    sequence[j] = S.pop( )
```

R-6.6) Hint Give a recursive definition.

R-6.6) Solution An arithmetic expression has matching grouping symbols if it has one of the following structures, where S denotes an arithmetic expression without grouping symbols, and E, E', and E'' recursively denote an arithmetic expression with matching grouping symbols:

- S
- $E'(E)E''$
- $E'[E]E''$
- $E'\{E\}E''$

R-6.7) Hint Use a paper and pencil with eraser to simulate the queue.

R-6.7) Solution 5, 3, 2, 8, 9, 1, 7, 6

R-6.8) Hint If a queue is empty when dequeue is called, its size does not change.

R-6.8) Solution The size of the queue is $32 - 15 + 5 = 22$.

R-6.9) Hint Each successful dequeue operation causes that index to shift circularly to the right.

R-6.9) Solution _front $= 10$.

R-6.10) Hint Consider how the queue might be configured within the underlying array.

R-6.11) Hint Read the documentation of collections.deque, if needed.

R-6.12) Hint Use a paper and pencil to simulate the deque.

R-6.12) Solution 9, 5, 9, 4, 7, 3, 4, 8

R-6.13) Hint You may use the return value of a removal method as a parameter to an insertion method.

R-6.13) Solution

```
D.add_last(D.delete_first( ))
D.add_last(D.delete_first( ))
D.add_last(D.delete_first( ))
Q.enqueue(D.delete_first( ))
Q.enqueue(D.delete_first( ))
D.add_first(Q.dequeue( ))
D.add_first(Q.dequeue( ))
D.add_first(D.delete_back( ))
D.add_first(D.delete_back( ))
D.add_first(D.delete_back( ))
```

R-6.14) Hint You may use the return value of a removal method as a parameter to an insertion method. Think about how to effectively use the stack for temporary storage.

R-6.14) Solution

```
D.add_last(D.delete_first( ))
D.add_last(D.delete_first( ))
D.add_last(D.delete_first( ))
D.add_last(D.delete_first( ))
S.push(D.delete_first( ))
D.add_first(D.delete_last( ))
D.add_first(S.pop( ))
D.add_first(D.delete_last( ))
D.add_first(D.delete_last( ))
D.add_first(D.delete_last( ))
```

Creativity

C-6.15) Hint Pop the top integer, but remember it.

C-6.15) Solution

```
x = S.pop( )
if x < S.top( ):
  x = S.pop( )
```

Note that if the largest integer is the first or second element of S, then x will store it. Thus, x stores the largest element with probability $2/3$.

C-6.16) Hint Use a new instance variable to store the capacity limit.

C-6.17) Hint Use an expression such as [**None**] $*$ k to build a list of k **None** values.

C-6.18) Hint You will need to do three transfers.

C-6.18) Solution

```
transfer(S, A)     # contents in A are in reverse order as original
transfer(A, B)     # contents in B are in same order as original
transfer(B, S)     # contents in S are in reverse order as original
```

C-6.19) Hint After finding what's between the $<$ and $>$ characters, the tag is only the part before the first space (if any).

C-6.20) Hint Use a stack to reduce the problem to that of enumerating all permutations of the numbers $\{1,2,\ldots,n-1\}$.

C-6.21) Hint Use the stack to store the elements yet to be used to generate subsets and use the queue to store the subsets generated so far.

C-6.22) Hint Use a stack.

C-6.23) Hint You can still use R as temporary storage, as long as you never pop its original contents.

C-6.23) Solution Let r, s and t denote the original sizes of the stacks.

Make s calls to R.push(S.pop( ))
Make t calls to R.push(T.pop( ))
Make $s+t$ calls to S.push(R.pop( ))

C-6.24) Hint Rotate elements within the queue.

C-6.24) Solution One approach to implement the stack ADT using a queue Q, simply enqueues elements into Q whenever a push call is made. This takes $O(1)$ time to complete. For pop calls, we can call Q.enqueue(Q.dequeue()) $n-1$ times, where n is the current size, and then return Q.dequeue(), as that is the most recently inserted element. This requires $O(n)$ time. We can use a similar approach for top, but rotating the answer back to the end of the queue.

A better approach is to align the elements of the queue so that the "top" is aligned with the front of the queue. This can be done by enqueuing a pushed element, and then doing $n-1$ immediate calls to Q.enqueue(Q.dequeue()). This requires $O(n)$ time, but with that orientation, both pop and top can be implemented in $O(1)$ time by a respective call to dequeue or front.

C-6.25) Hint Consider using one stack to collect incoming elements, and another as a buffer for elements to be delivered.

C-6.25) Solution

Consider the following implementation.

```
class SolnQueue:
  def __init__(self):
    self._incoming = new Stack( )
    self._outgoing = new Stack( )

  def _transfer(self):
    while not self.incoming.is_empty( ):
      self.outgoing.push(self.incoming.pop( ))

  def __len__(self):
    return len(self.incoming) + len(self.outgoing)

  def is_empty(self):
    return self.incoming.is_empty( ) and self.outgoing.is_empty( )

  def enqueue(self, e):
    self.incoming.push(e)

  def front(self):
    if self.is_empty( ):
      raise Empty('Queue is empty')
    if self.outgoing.is_empty( ):
      self._transfer( )
    return self.outgoing.top( )

  def dequeue(self):
    if self.is_empty( ):
      raise Empty('Queue is empty')
    if self.outgoing.is_empty( ):
      self._transfer( )
    return self.outgoing.pop( )
```

The correctness can be proven by arguing that elements in the outgoing stack are ordered with the older elements toward the top, and that all elements in the outgoing stack are older than any elements in the incoming stack.

Although the worst-case running time for front and dequeue are $O(n)$ due to a transfer, the $O(1)$ amortized bound can be seen from the fact that any given element participates in at most one transfer in its lifetime, and thus the total time spent in the transfer loop during a sequence of k queue operations is k.

C-6.26) Hint Think of using one stack for each end of the deque.

C-6.27) Hint Think of how you might use Q to process the elements of S twice.

C-6.27) Solution The solution is to actually use the queue Q to process the elements in two phases. In the first phase, we iteratively pop each the element from S and enqueue it in Q, and then we iteratively dequeue each element from Q and push it into S. This reverses the elements in S. Then we repeat this same process, but this time we also look for the element x. By passing the elements through Q and back to S a second time, we reverse the reversal, thereby putting the elements back into S in their original order.

C-6.28) Hint Use a new instance variable to store the capacity limit.

C-6.29) Hint You might start by combining the code of a dequeue followed by an enqueue, and then simplify.

C-6.29) Solution

```
def rotate(self):
  if not self.is_empty( ):
    temp = self._data[self._front]
    self._data[self._front] = None
    self._front = (self._front + 1) % len(self._data)
    avail = (self._front + self._size) % len(self._data)
    self._data[avail] = temp
```

C-6.30) Hint Think of the queues like boxes and the integers like red and blue marbles.

C-6.30) Solution Alice should put an even integer in Q and all the other 99 integers in R. This gives her a $74/99$ (roughly 74.7%) chance of winning.

C-6.31) Hint Lazy and Crazy should only go across once.

C-6.31) Solution Mazie and Daisy go across (4 min.), Daisy comes back (4 min.), Crazy and Lazy go across (20 min.), Mazie comes back (2 min.), and then Mazie and Daisy go across (4 min.).

Projects

P-6.32) Hint Suggested instance variables are described in the book.

P-6.33) Hint What is the index of a new element? What is the index of the old element that is lost?

P-6.34) Hint You will need to use a stack.

P-6.35) Hint How does this functionality compare to a deque?

P-6.36) Hint Keep information about the purchase shares and prices in a queue, and then match those against sales. Care must be taken if only part of a purchase block is sold.

P-6.37) Hint Start one stack at each end of the array, growing toward the center.