CHAPTER 8

Queues

The term queue is commonly defined to be a line of people waiting to be served like those you would encounter at many business establishments. Each person is served based on their position within the queue. Thus, the next person to be served is the first in line. As more people arrive, they enter the queue at the back and wait their turn.

A queue structure is well suited for problems in computer science that require data to be processed in the order in which it was received. Some common examples include computer simulations, CPU process scheduling, and shared printer management. You are familiar with a printer queue if you have used a shared printer. Many people may want to use the printer, but only one thing can be printed at a time. Instead of making people wait until the printer is not being used to print their document, multiple documents can be submitted at the same time. When a document arrives, it is added to the end of the print queue. As the printer becomes available, the document at the front of the queue is removed and printed.

8.1 The Queue ADT

A ***queue*** is a specialized list with a limited number of operations in which items can only be added to one end and removed from the other. A queue is also known as a *first-in, first-out* (FIFO) list. Consider Figure 8.1, which illustrates an abstract view of a queue. New items are inserted into a queue at the *back* while existing items are removed from the *front*. Even though the illustration shows the individual items, they cannot be accessed directly. The definition of the Queue ADT follows.

Figure 8.1: An abstract view of a queue containing five items.

Define **Queue ADT**

A *queue* is a data structure that a linear collection of items in which access is restricted to a first-in first-out basis. New items are inserted at the back and existing items are removed from the front. The items are maintained in the order in which they are added to the structure.

- `Queue()`: Creates a new empty queue, which is a queue containing no items.
- `isEmpty()`: Returns a boolean value indicating whether the queue is empty.
- *`length`*`()`: Returns the number of items currently in the queue.
- `enqueue(item)`: Adds the given item to the back of the queue.
- `dequeue()`: Removes and returns the front item from the queue. An item cannot be dequeued from an empty queue.

Using the formal definition of the Queue ADT, we can now examine the code necessary to create the queue in Figure 8.1:

```
Q = Queue()
Q.enqueue( 28 )
Q.enqueue( 19 )
Q.enqueue( 45 )
Q.enqueue( 13 )
Q.enqueue( 7 )
```

After creating a `Queue` object, we simply enqueue the five values in the order as they appear in the queue. We can then remove the values or add additional values to the queue. Figure 8.2 illustrates the result of performing several additional operations on the sample queue.

8.2 Implementing the Queue

Since the queue data structure is simply a specialized list, it is commonly implemented using some type of list structure. There are three common approaches to implementing a queue: using a vector, a linked list, or an array. In the following sections, we examine and compare these three approaches.

8.2.1 Using a Python List

The simplest way to implement the Queue ADT is to use Python's list. It provides the necessary routines for adding and removing items at the respective ends. By applying these routines, we can remove items from the front of the list and append new items to the end. To use a list for the Queue ADT, the constructor must define a single data field to store the list that is initially empty. We can test for an

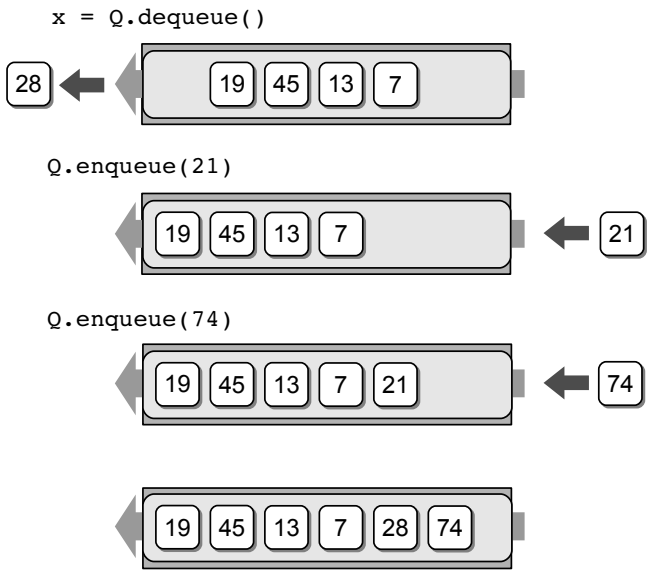


Figure 8.2: Abstract view of the queue after performing additional operations.

empty queue by examining the length of the list. The complete Python list-based implementation is provided in Listing 8.1, and an instance of the class is illustrated in Figure 8.3 on the next page.

To enqueue an item, we simply append it to the end of the list. The dequeue operation can be implemented by popping and returning the item in the first

Listing 8.1 The `pylistqueue.py` module.

```
1  # Implementation of the Queue ADT using a Python list.
2  class Queue :
3     # Creates an empty queue.
4    def __init__( self ):
5      self._qList = list()
6
7     # Returns True if the queue is empty.
8    def isEmpty( self ):
9      return len( self ) == 0
10
11    # Returns the number of items in the queue.
12   def __len__( self ):
13     return len( self._qList )
14
15    # Adds the given item to the queue.
16   def enqueue( self, item ):
17     self._qList.append( item )
18
19    # Removes and returns the first item in the queue.
20   def dequeue( self ):
21     assert not self.isEmpty(), "Cannot dequeue from an empty queue."
22     return self._qList.pop( 0 )
```

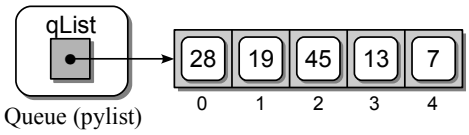


Figure 8.3: An instance of the Queue ADT implemented using a Python list.

element of the list. Before attempting to remove an item from the list, we must ensure the queue is not empty. Remember, the queue definition prohibits the use of the `dequeue()` operation on an empty queue. Thus, to enforce this, we must first assert the queue is not empty and raise an exception, when the operation is attempted on an empty queue.

Since we use list operations to implement the individual queue operations, we need only recall the worst case times for the Python list operations. The size and empty condition operations only require $O(1)$ time. The enqueue operation requires $O(n)$ time in the worst case since the list may need to expand to accommodate the new item. When used in sequence, the enqueue operation has an amortized cost of $O(1)$. The dequeue operation also requires $O(n)$ time since the underlying array used to implement the Python list may need to shrink when an item is removed. In addition, when an item is removed from the front of the list, the following items have to be shifted forward, which requires linear time no matter if an expansion occurs or not.

8.2.2 Using a Circular Array

The list-based implementation of the Queue ADT is easy to implement, but it requires linear time for the enqueue and dequeue operations. We can improve these times using an array structure and treating it as a circular array. A ***circular array*** is simply an array viewed as a circle instead of a line. An example is illustrated in Figure 8.4.

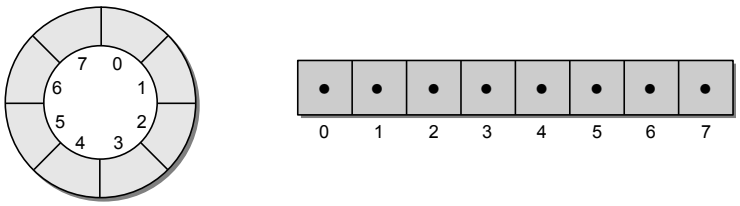
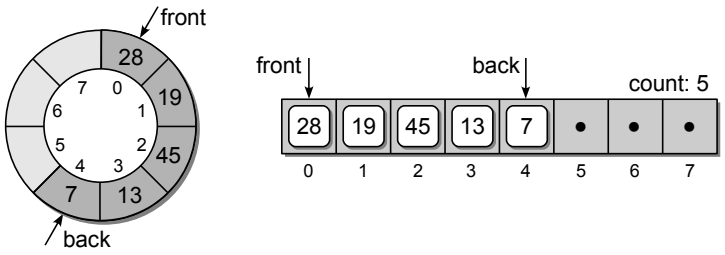


Figure 8.4: The abstract view of a circular array (left) and the physical view (right).

A circular array allows us to add new items to a queue and remove existing ones without having to shift items in the process. Unfortunately, this approach introduces the concept of a maximum-capacity queue that can become full. A circular array queue implementation is typically used with applications that only require small-capacity queues and allows for the specification of a maximum size.

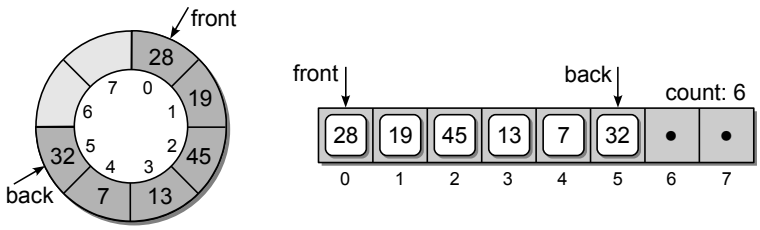
Data Organization

To implement a queue as a circular array, we must maintain a count field and two markers. The count field is necessary to keep track of how many items are currently in the queue since only a portion of the array may actually contain queue items. The markers indicate the array elements containing the first and last items in the queue. Consider the following circular array:

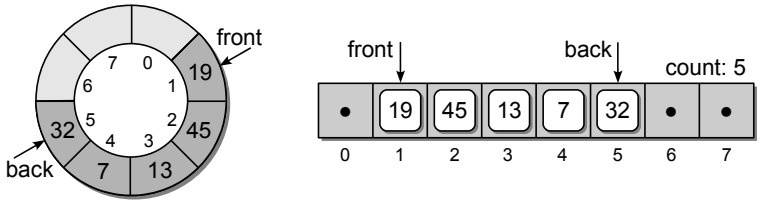


which illustrates the implementation of the queue from Figure 8.1. The figure shows the corresponding abstract and physical views of the circular array.

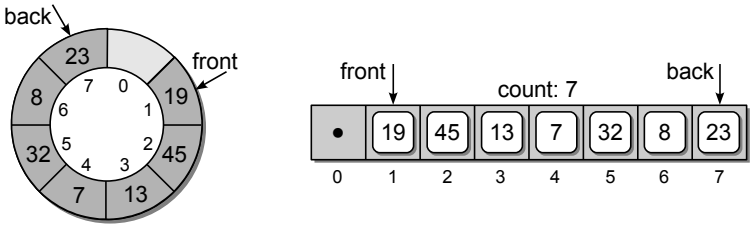
New items are added to the queue by inserting them in the position immediately following the `back` marker. The marker is then advanced one position and the counter is incremented to reflect the addition of the new item. For example, suppose we enqueue value 32 into the sample queue. The `back` marker is advanced to position 5 and value 32 is inserted:



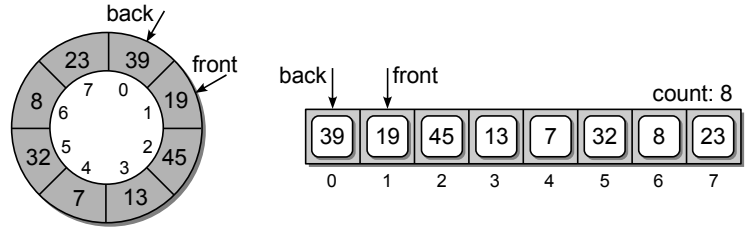
To dequeue an item, the value in the element marked by `front` will be returned and the marker is advanced one position:



Notice the remaining items in the queue are not shifted. Instead, only the `front` marker is moved. Now, suppose we add values 8 and 23 to the queue. These values are added in the positions following the `back` marker:



The queue now contains seven items in elements [1 . . . 7] with one empty slot. What happens if value 39 is added? Since we are using a circular array, the same procedure is used and the new item will be inserted into the position immediately following the `back` marker. In this case, that position will be element 0. Thus, the queue wraps around the circular array as items are added and removed, which eliminates the need to shift items. The resulting queue is shown here:



This also represents a full queue since all slots in the array are filled. No additional items can be added until existing items have been removed. This is a change from the original definition of the Queue ADT and requires an additional operation to test for a full queue.

Queue Implementation

Given the description of a circular array and its use in implementing a queue, we turn our attention to the implementation details. A Python implementation of the Queue ADT using a circular array is provided in Listing 8.2.

The constructor creates an object containing four data fields, including the counter to keep track of the number of items in the queue, the two markers, and the array itself. A sample instance of the class is illustrated in Figure 8.5.

For the circular queue, the array is created with `maxSize` elements as specified by the argument to the constructor. The two markers are initialized so the first item will be stored in element 0. This is achieved by setting `_front` to 0 and `_back` to the index of the last element in the array. When the first item is added, `_back`

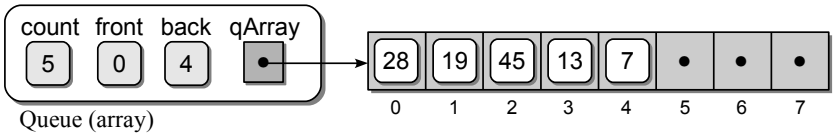


Figure 8.5: A Queue object implemented as a circular array.

Listing 8.2 The `arrayqueue.py` module.

```
1  # Implementation of the Queue ADT using a circular array.
2  from array import Array
3
4  class Queue :
5      # Creates an empty queue.
6    def __init__( self, maxSize ) :
7      self._count = 0
8      self._front = 0
9      self._back = maxSize - 1
10     self._qArray = Array( maxSize )
11
12    # Returns True if the queue is empty.
13   def isEmpty( self ) :
14     return self._count == 0
15
16    # Returns True if the queue is full.
17   def isFull( self ) :
18     return self._count == len(self._qArray)
19
20    # Returns the number of items in the queue.
21   def __len__( self ) :
22     return self._count
23
24    # Adds the given item to the queue.
25   def enqueue( self, item ):
26     assert not self.isFull(), "Cannot enqueue to a full queue."
27     maxSize = len(self._qArray)
28     self._back = (self._back + 1) % maxSize
29     self._qArray[self._back] = item
30     self._count += 1
31
32    # Removes and returns the first item in the queue.
33   def dequeue( self ):
34     assert not self.isEmpty(), "Cannot dequeue from an empty queue."
35     item = self._qArray[ self._front ]
36     maxSize = len(self._qArray)
37     self._front = (self._front + 1) % maxSize
38     self._count -= 1
39     return item
```

will wrap around to element 0 and the new value will be stored in that position. Figure 8.6 illustrates the circular array when first created by the constructor.

The `size()` and `isEmpty()` methods use the value of `_count` to return the appropriate result. As indicated earlier, implementing the Queue ADT as a circular array creates the special case of a queue with a maximum capacity, which can result in a full queue. For this implementation of the queue, we must add the `isFull()` method, which can be used to test if the queue is full. Again, the `_count` field is used to determine when the queue becomes full.

To enqueue an item, as shown in lines 25–30, we must first test the precondition and verify the queue is not full. If the condition is met, the new item can be inserted

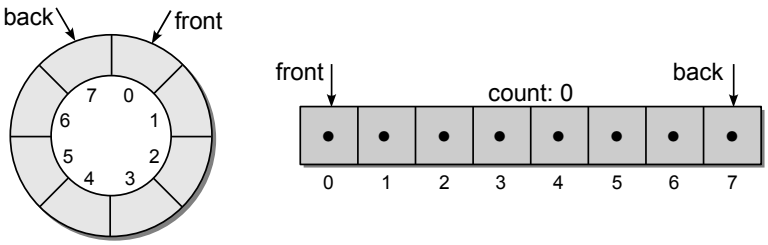


Figure 8.6: The circular array when the queue is first created in the constructor.

into the position immediately following the `_back` marker. But remember, we are using a circular array and once the marker reaches the last element of the actual linear array, it must wrap around to the first element. This can be done using a condition statement to test if `_back` is referencing the last element and adjusting it appropriately, as shown here:

```
self._back += 1
if self._back == len( self._qArray ) :
  self._back = 0
```

A simpler approach is to use the modulus operator as part of the increment step. This reduces the need for the conditional and automatically wraps the marker to the beginning of the array as follows:

```
self._back = ( self._back + 1 ) % len( self._qArray )
```

The dequeue operation is implemented in a similar fashion as `enqueue()` as shown in lines 33–39 of Listing 8.2. The item to be removed is taken from the position marked by `_front` and saved. The marker is then advanced using the modulus operator as was done when enqueueing a new item. The counter is decremented and the saved item is returned.

Run Time Analysis

The circular array implementation provides a more efficient solution than the Python list. The operations all have a worst case time-complexity of $O(1)$ since the array items never have to be shifted. But the circular array does introduce the drawback of working with a maximum-capacity queue. Even with this limitation, it is well suited for some applications.

8.2.3 Using a Linked List

A major disadvantage in using a Python list to implement the Queue ADT is the expense of the enqueue and dequeue operations. The circular array implementation improved on these operations, but at the cost of limiting the size of the queue. A better solution is to use a linked list consisting of both head and tail references. Adding the tail reference allows for quick append operations that otherwise would

require a complete traversal to find the end of the list. Figure 8.7 illustrates a sample linked list with the two external references.

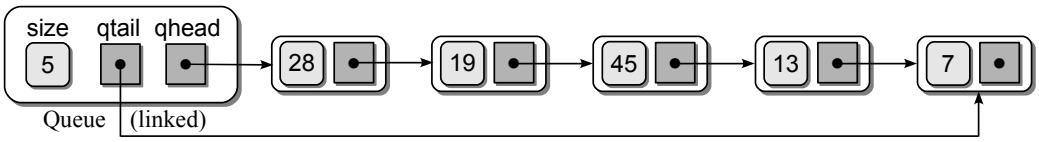


Figure 8.7: An instance of the Queue ADT implemented as a linked list.

The complete implementation of the Queue ADT using a linked list with a tail reference is provided in Listing 8.3. Remember, the individual nodes in the list contain the individual items in the queue. When dequeueing an item, we must unlink the node from the list but return the item stored in that node and not the node itself. An evaluation of the time-complexities is left as an exercise.

Listing 8.3 The `llistqueue.py` module.

```
1  # Implementation of the Queue ADT using a linked list.
2  class Queue :
3      # Creates an empty queue.
4    def __init__( self ):
5      self._qhead = None
6      self._qtail = None
7      self._count = 0
8
9     # Returns True if the queue is empty.
10   def isEmpty( self ):
11     return self._qhead is None
12
13    # Returns the number of items in the queue.
14   def __len__( self ):
15     return self._count
16
17    # Adds the given item to the queue.
18   def enqueue( self, item ):
19     node = _QueueNode( item )
20     if self.isEmpty() :
21       self._qhead = node
22     else :
23       self._qtail.next = node
24
25     self._qtail = node
26     self._count += 1
27
28    # Removes and returns the first item in the queue.
29   def dequeue( self ):
30     assert not self.isEmpty(), "Cannot dequeue from an empty queue."
31     node = self._qhead
32     if self._qhead is self._qtail :
33       self._qtail = None
```

(Listing Continued)

Listing 8.3 Continued ...

```
34
35      self._qhead = self._qhead.next
36      self._count -= 1
37      return node.item
38
39  # Private storage class for creating the linked list nodes.
40  class _QueueNode( object ):
41    def __init__( self, item ):
42      self.item = item
43      self.next = None
```

8.3 Priority Queues

Some applications require the use of a queue in which items are assigned a priority and the items with a higher priority are dequeued first. However, all items with the same priority still obey the FIFO principle. That is, if two items with the same priority are enqueued, the first in will be the first out.

8.3.1 The Priority Queue ADT

A priority queue is simply an extended version of the basic queue with the exception that a priority p must be assigned to each item at the time it is enqueued. There are two basic types of priority queues: bounded and unbounded. The ***bounded priority queue*** assumes a small limited range of p priorities over the interval of integers $[0 \ldots p)$. The ***unbounded priority queue*** places no limit on the range of integer values that can be used as priorities. The definition of the Priority Queue ADT is provided below. Note that we use one definition for both bounded and unbounded priority queues.

Define **Priority Queue ADT**

A *priority queue* is a queue in which each item is assigned a priority and items with a higher priority are removed before those with a lower priority, irrespective of when they were added. Integer values are used for the priorities with a smaller integer value having a higher priority. A *bounded priority queue* restricts the priorities to the integer values between zero and a predefined upper limit whereas an *unbounded priority queue* places no limits on the range of priorities.

- `PriorityQueue()`: Creates a new empty unbounded priority queue.
- `BPriorityQueue( numLevels )`: Creates a new empty bounded priority queue with priority levels in the range from 0 to `numLevels - 1`.
- `isEmpty()`: Returns a boolean value indicating whether the queue is empty.

- *length*(): Returns the number of items currently in the queue.
- enqueue(item, priority): Adds the given item to the queue by inserting it in the proper position based on the given priority. The priority value must be within the legal range when using a bounded priority queue.
- dequeue(): Removes and returns the front item from the queue, which is the item with the highest priority. The associated priority value is discarded. If two items have the same priority, then those items are removed in a FIFO order. An item cannot be dequeued from an empty queue.

Consider the following code segment, which enqueues a number of items into a priority queue. The priority queue is defined with six levels of priority with a range of $[0 \ldots 5]$. The resulting queue is shown in Figure 8.8.

```
Q = BPriorityQueue( 6 )
Q.enqueue( "purple", 5 )
Q.enqueue( "black", 1 )
Q.enqueue( "orange", 3 )
Q.enqueue( "white", 0 )
Q.enqueue( "green", 1 )
Q.enqueue( "yellow", 5 )
```

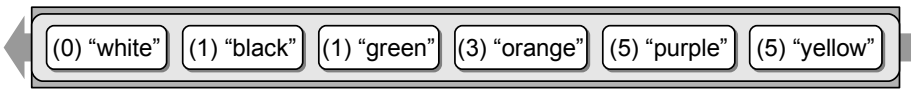


Figure 8.8: Abstract view of a priority queue resulting from enqueueing several strings, along with individual priorities.

The first item to be removed will be the first item with the highest priority. Notice when items "*black*" and "*green*" are enqueued, "*green*" follows "*black*" in the queue even though they have the same priority since items with equal priority still obey the FIFO principle. The following code segment removes the items and prints them to the terminal:

```
while not Q.isEmpty() :
  item = Q.dequeue()
  print( item )
```

which results in the following output:

```
white
black
green
orange
purple
yellow
```

8.3.2 Implementation: Unbounded Priority Queue

There are a number of ways to implement an unbounded Priority Queue ADT. The most basic is to use a Python list or linked list as was done with the Queue ADT. To implement the priority queue, we must consider several facts related to the definition of the ADT:

- A priority must be associated with each item in the queue, possibly requiring the value to be stored along with the item.
- The next item to be dequeued from the priority queue is the item with the highest priority.
- If multiple items have the same priority, those items must be dequeued in the order they were originally enqueued.

Python List Implementation

We used a Python list to implement the basic queue earlier in the chapter. In that implementation, the queue items were organized within the list from front to back with new items appended directly to the end and existing items removed from the front. That simple organization worked well with the basic queue. When implementing the priority queue, however, the items cannot simply be added directly to the list, but instead we must have a way to associate a priority with each item. This can be accomplished with a simple storage class containing two fields: one for the priority and one for the queue item. For example:

```
class _PriorityQEntry :
  def __init__( self, item, priority ):
    self.item = item
    self.priority = priority
```

With the use of a storage class for maintaining the associated priorities, the next question is how should the entries be organized within the vector? We can consider two approaches, both of which satisfy the requirements of the priority queue:

- *Append new items to the end of the list.* When a new item is enqueued, simply append a new instance of the storage class (containing the item and its priority) to the end of the list. When an item is dequeued, search the vector for the item with the lowest priority and remove it from the list. If more than one item has the same priority, the first one encountered during the search will be the first to be dequeued.
- *Keep the items sorted within the list based on their priority.* When a new item is enqueued, find its proper position within the list based on its priority and insert an instance of the storage class at that point. If we order the items in the vector from lowest priority at the front to highest at the end, then the dequeue operation simply requires the removal of the last item in the list. To maintain

the proper ordering of items with equal priority, the enqueue operation must ensure newer items are inserted closer to the front of the list than the other items with the same priority.

An implementation of the priority queue using a Python list in which new items are appended to the end is provided in Listing 8.4. A sample instance of the class is illustrated in Figure 8.9. Note this implementation does not use the `numLevels` argument passed to the constructor since we can store items having any number of priority levels.

Listing 8.4 The `priorityq.py` module.

```
# Implementation of the unbounded Priority Queue ADT using a Python list
# with new items appended to the end.

class PriorityQueue :
   # Create an empty unbounded priority queue.
  def __init__( self ):
    self._qList = list()

   # Returns True if the queue is empty.
  def isEmpty( self ):
    return len( self ) == 0

   # Returns the number of items in the queue.
  def __len__( self ):
    return len( self._qList )

   # Adds the given item to the queue.
  def enqueue( self, item, priority ):
     # Create a new instance of the storage class and append it to the list.
    entry = _PriorityQEntry( item, priority )
    self._qList.append( entry )

   # Removes and returns the first item in the queue.
  def dequeue( self ) :
    assert not self.isEmpty(), "Cannot dequeue from an empty queue."

     # Find the entry with the highest priority.
    highest = self._qList[i].priority
    for i in range( self.len() ) :
       # See if the ith entry contains a higher priority (smaller integer).
      if self._qList[i].priority < highest :
        highest = self._qList[i].priority

     # Remove the entry with the highest priority and return the item.
    entry = self._qList.pop( highest )
    return entry.item

# Private storage class for associating queue items with their priority.
class _PriorityQEntry( object ):
  def __init__( self, item, prioity ):
    self.item = item
    self.priority = priority
```

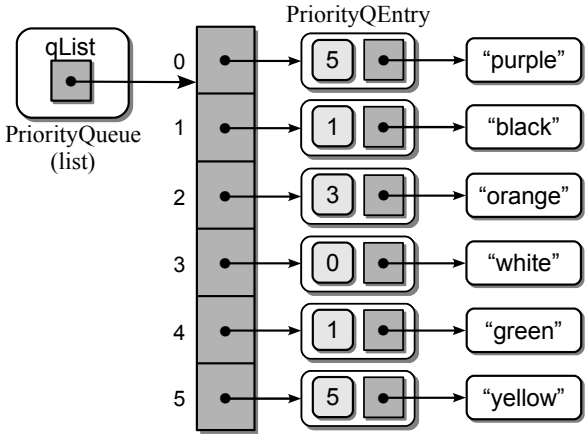


Figure 8.9: An instance of the priority queue implemented using a list.

To evaluate the efficiency, we consider the implementation of each operation. Testing for an empty queue and determining the size can be done in $O(1)$ time. The enqueue operation requires linear time in the worst case since the underlying array may have to be reallocated, but it has a $O(1)$ amortized cost. The dequeue operation is also $O(n)$ since we must search through the entire list in the worst case to find the entry with the highest priority.

Linked List Implementation

A singly linked list structure with both head and tail references can also be used to implement the priority queue as illustrated in Figure 8.10. The `_QueueNode` class used in the implementation of the Queue ADT using a linked list has to be modified to include the priority value. After that, the linked list can be used in a similar fashion as the Python list. When an item is enqueued, the new node is appended to the end of the linked list and when a dequeue operation is performed, the linked list is searched to find the entry with the highest priority. We leave the actual design and implementation as an exercise but examine the time-complexity of this approach.

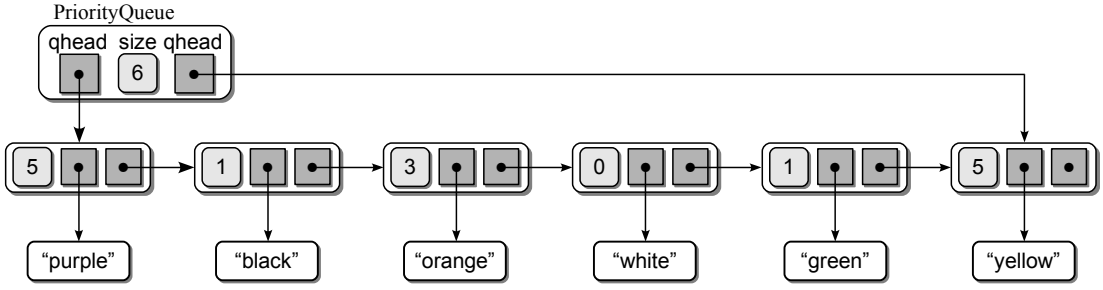


Figure 8.10: Implementation of the priority queue from Figure 8.8 using a linked list.

As with the Python list implementation of the priority queue, testing for an empty queue and determining the size can be done in $O(1)$ time. The enqueue operation can also be done in constant time since we need only append a new node to the end of the list. The dequeue operation, however, requires $O(n)$ time since the entire list must be searched in the worst case to find the entry with the highest priority. Once that entry is located, the node can be removed from the list in constant time.

8.3.3 Implementation: Bounded Priority Queue

The Python list and linked list versions are quite simple to implement and can be used to implement the bounded priority queue. But they both require linear time to dequeue an item. Since the priorities of a bounded priority queue are restricted to a finite set $[0 \ldots p)$, we can improve this worst case time with an implementation in which all of the operations only require constant time. This can be done using an array of queues, as illustrated in Figure 8.11.

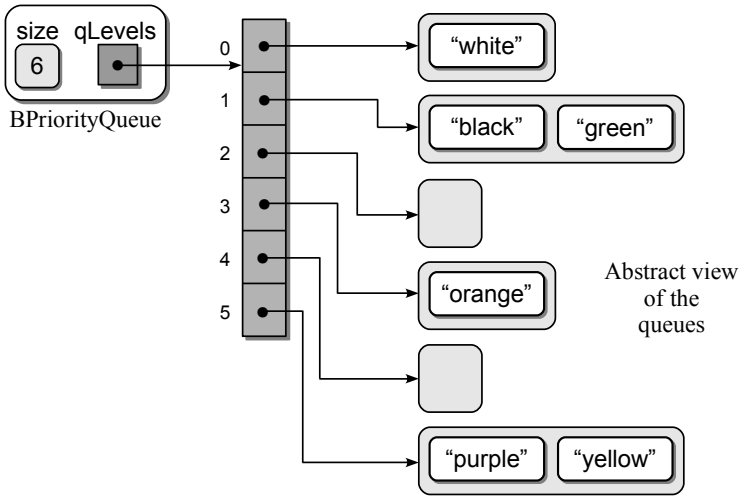


Figure 8.11: Implementation of the priority queue using an array of queues.

The implementation of the bounded Priority Queue ADT is shown in Listing 8.5. The constructor creates two data fields. The `_qLevels` field contains an array of p elements in which each contains an instance of the Queue ADT. The `_size` field maintains the number of items in the priority queue, which can also be used to determine if the queue is empty. When an item with priority k is added to the priority queue, it is added to the queue stored in the `qList[k]` element. To dequeue an item from the priority queue, we must iterate through the array to find the first non-empty queue, which will contain the first item to be removed from the priority queue. Note that we do not have to store the priorities along with each item since all items with a given priority will be stored in the same queue. The priority can be determined from the array indices.

The implementation of the priority queue using an array of queues is also quite simple. But can we obtain constant time operations? We begin with the `isEmpty()` and `__len__` operations. Since a data field is maintained to store the number of items in the priority queue, both can be performed in constant time. The enqueue operation also requires constant time since adding an item to a general queue can be done in constant time as can accessing an individual queue in the array. Dequeueing from a general queue implemented as a linked list can be done in constant time. But since we must iterate through the array to find the first non-empty queue, the priority dequeue operation requires $O(p)$ time. While this time is linear, it is linear with respect to the number of priorities and not to the number of elements in the queue (n). When the number of priorities is quite small,

Listing 8.5 The `bpriorityq.py` module.

```
# Implementation of the bounded Priority Queue ADT using an array of
# queues in which the queues are implemented using a linked list.
from array import Array
from llistqueue import Queue

class BPriorityQueue :
   # Creates an empty bounded priority queue.
  def __init__( self, numLevels ):
    self._qSize = 0
    self._qLevels = Array( numLevels )
    for i in range( numLevels ) :
      self._qLevels[i] = Queue()

   # Returns True if the queue is empty.
  def isEmpty( self ):
    return len( self ) == 0

   # Returns the number of items in the queue.
  def __len__( self ):
    return len( self._qSize )

   # Adds the given item to the queue.
  def enqueue( self, item, priority ):
    assert priority >= 0 and priority < len(self._qLevels), \
           "Invalid priority level."
    self._qLevels[priority].enqueue( item )

   # Removes and returns the next item in the queue.
  def dequeue( self ) :
     # Make sure the queue is not empty.
    assert not self.isEmpty(), "Cannot dequeue from an empty queue."
     # Find the first non-empty queue.
    i = 0
    p = len(self._qLevels)
    while i < p and not self._qLevels[i].isEmpty() :
      i += 1
     # We know the queue is not empty, so dequeue from the ith queue.
    return self._qLevels[i].dequeue()
```

we can safely treat p as a constant value and specify the dequeue operation as requiring constant time.

The disadvantage of this structure for the implementation of the priority queue is that the number of levels is fixed. If an application requires a priority queue with an unlimited number of priority levels, then the vector or linked-list versions are a better choice.

8.4 Application: Computer Simulations

Computers have long been used to model and simulate real-world systems and phenomena. These simulations are simply computer applications that have been designed to represent and appropriately react to the significant events occurring in the system. Simulations can allow humans to study certain behaviors or experiment with certain changes and events in a system to determine the appropriate strategy.

Some of the more common simulations include weather forecasting and flight simulators. A flight simulator, which is a mock-up of a real cockpit and controlled by software, helps train pilots to deal with real situations without having to risk the life of the pilot or loss of the aircraft. Weather forecasts today are much more reliable due to the aide of computer simulations. Mathematical models have been developed to simulate weather patterns and atmospheric conditions. These models can be solved using computer applications, which then provide information to meteorologists for use in predicting the weather.

Computer simulations are also used for less glamorous applications. Businesses can use a computer simulation to determine the number of employees needed to provide a service to its customers. For example, an airline may want to know how many ticket agents are needed at certain times of the day in order to provide timely service. Having too many agents will cost the airline money, but too few will result in angry customers. The company could simply study the customer habits at one airport and experiment with a different number of agents at different times. But this can be costly and time consuming. In addition, the results may only be valid for that one airport. To reduce the cost and allow for events that may occur at various airports, a computer simulation can be developed to model the real system.

8.4.1 Airline Ticket Counter

Simulating an airline ticket counter, or any other ***queuing system*** where customers stand in line awaiting service, is very common. A queue structure is used to model the queuing system in order to study certain behaviors or outcomes. Some of the typical results studied include average waiting time and average queue length. Queuing systems that use a single queue are easier to model. More complex systems like those representing a grocery store that use multiple queues, require more complex models. In this text, we limit our discussion to single-queue systems.

Queuing System Model

We can model a queuing system by constructing a ***discrete event simulation***. The simulation is a sequence of significant events that cause a change in the system. For example, in our airline ticket counter simulation, these events would include customer arrival, the start or conclusion of a transaction, or customer departure.

The simulation is time driven and performed over a preset time period. The passing of time is represented by a loop, which increments a discrete time variable once for each tick of the clock. The events can only occur at discrete time intervals. Thus, the time units must be small enough such that no event can occur between units. A simulation is commonly designed to allow the user to supply parameters that define the conditions of the system. For a discrete event simulation modeling a queuing system, these parameters include:

- The length of the simulation given in number of time units. The simulation typically begins at time unit zero.
- The number of servers providing the service to the customers. We must have at least one server.
- The expected service time to complete a transaction.
- The distribution of arrival times, which is used to determine when customers arrive.

By adjusting these parameters, the user can change the conditions under which the simulation is performed. We can change the number of servers, for example, to determine the optimal number required to provide satisfactory service under the given conditions.

Finally, a set of rules are defined for handling the events during each tick of the clock. The specific rules depends on what results are being studied. To determine the average time customers must wait in line before being served, there are three rules:

Rule 1: If a customer arrives, he is added to the queue. At most, one customer can arrive during each time step.

Rule 2: If there are customers waiting, for each free server, the next customer in line begins her transaction.

Rule 3: For each server handling a transaction, if the transaction is complete, the customer departs and the server becomes free.

When the simulation completes, the average waiting time can be computed by dividing the total waiting time for all customers by the total number of customers.

Random Events

To correctly model a queuing system, some events must occur at random. One such event is customer arrival. In the first rule outlined earlier, we need to determine if

a customer arrives during the current tick of the clock. In a real-world system, this event cannot be directly controlled but is a true random act. We need to model this action as close as possible in our simulation.

A simple approach would be to flip a coin and let "heads" represent a customer arrival. But this would indicate that there is a 50/50 chance a customer arrives every time unit. This may be true for some systems, but not necessarily the one we are modeling. We could change and use a six-sided die and let one of the sides represent a customer arrival. But this only changes the odds to 1 in 6 that a customer arrives.

A better approach is to allow the user to specify the odds of a customer arriving at each time step. This can be done in one of two ways. The user can enter the odds a customer arrives during the current time step as a real value between 0.0 (no chance) and 1.0 (a sure thing). If 0.2 is entered, then this would indicate there is a 1 in 5 chance a customer arrives. Instead of directly entering the odds, we can have the user enter the average time between customer arrivals. We then compute the odds within the program. If the user enters an average time of 8.0, then on average a customer arrives every 8 minutes. But customers can arrive during any minute of the simulation. The average time between arrivals simply provides the average over the entire simulation. We use the average time to compute the odds of a customer arriving as 1.0/8.0, or 0.125.

Given the odds either directly by the user or computing them based on the average arrival time, how is this value used to simulate the random act of a customer arriving? We use the *random number generator* provided by Python to generate a number between 0.0 and 1.0. We compare this result to the probability (`prob`) of an arrival. If the generated random number is between 0.0 and `prob` inclusive, the event occurs and we signal a customer arrival. On the other hand, if the random value is greater than `prob`, then no customer arrives during the current time step and no action is taken. The arrival probability can be changed to alter the number of customers served in the simulation.

8.4.2 Implementation

We can design and implement a discrete event computer simulation to analyze the average time passengers have to wait for service at an airport ticket counter. The simulation will involve multiple ticket agents serving customers who have to wait in line until they can be served. Our design will be an object-oriented solution with multiple classes.

System Parameters

The program will prompt the user for the queuing system parameters:

```
Number of minutes to simulate: 25
Number of ticket agents: 2
Average service time per passenger: 3
Average time between passenger arrival: 2
```

For simplicity we use minutes as the discrete time units. This would not be sufficient to simulate a real ticket counter as multiple passengers are likely to arrive within any given minute. The program will then perform the simulation and produce the following output:

```
Number of passengers served =  12
Number of passengers remaining in line = 1
The average wait time was 1.17 minutes.
```

We will also have the program display event information, which can be used to help debug the program. The debug information lists each event that occurs in the system along with the time the events occur. For the input values shown above, the event information displayed will be:

```
Time   2: Passenger 1 arrived.
Time   2: Agent 1 started serving passenger 1.
Time   3: Passenger 2 arrived.
Time   3: Agent 2 started serving passenger 2.
Time   5: Passenger 3 arrived.
Time   5: Agent 1 stopped serving passenger 1.
Time   6: Agent 1 started serving passenger 3.
Time   6: Agent 2 stopped serving passenger 2.
Time   8: Passenger 4 arrived.
Time   8: Agent 2 started serving passenger 4.
Time   9: Agent 1 stopped serving passenger 3.
Time  10: Passenger 5 arrived.
Time  10: Agent 1 started serving passenger 5.
Time  11: Passenger 6 arrived.
Time  11: Agent 2 stopped serving passenger 4.
Time  12: Agent 2 started serving passenger 6.
Time  13: Passenger 7 arrived.
Time  13: Agent 1 stopped serving passenger 5.
Time  14: Passenger 8 arrived.
Time  14: Agent 1 started serving passenger 7.
Time  15: Passenger 9 arrived.
Time  15: Agent 2 stopped serving passenger 6.
Time  16: Agent 2 started serving passenger 8.
Time  17: Agent 1 stopped serving passenger 7.
Time  18: Passenger 10 arrived.
Time  18: Agent 1 started serving passenger 9.
Time  19: Passenger 11 arrived.
Time  19: Agent 2 stopped serving passenger 8.
Time  20: Agent 2 started serving passenger 10.
Time  21: Agent 1 stopped serving passenger 9.
Time  22: Agent 1 started serving passenger 11.
Time  23: Passenger 12 arrived.
Time  23: Agent 2 stopped serving passenger 10.
Time  24: Agent 2 started serving passenger 12.
Time  25: Passenger 13 arrived.
Time  25: Agent 1 stopped serving passenger 11.
```

Passenger Class

First, we need a class to store information related to a single passenger. We create a `Passenger` class for this purpose. The complete implementation of this class is provided in Listing 8.6. The class will contain two data fields. The first is an identification number used in the output of the event information. The second field records the time the passenger arrives. This value will be needed to determine the length of time the passenger waited in line before beginning service with an agent. Methods are also provided to access the two data fields. An instance of the class is illustrated in Figure 8.12.

Listing 8.6 The `Passenger` class defined in the `simpeople.py` module.

```
1  # Used to store and manage information related to an airline passenger.
2  class Passenger :
3     # Creates a passenger object.
4    def __init__( self, idNum, arrivalTime ):
5      self._idNum = idNum
6      self._arrivalTime = arrivalTime
7
8     # Gets the passenger's id number.
9    def idNum( self ) :
10     return self._idNum
11
12    # Gets the passenger's arrival time.
13   def timeArrived( self ) :
14     return self._arrivalTime
```

Ticket Agent Class

We also need a class to represent and store information related to the ticket agents. The information includes an agent identification number and a timer to know when the transaction will be completed. This value is the sum of the current time and the average time of the transaction as entered by the user. Finally, we need to keep track of the current passenger being served by the agent in order to identify the specific passenger when the transaction is completed. The `TicketAgent` class is implemented in Listing 8.7, and an instance of the class is shown in Figure 8.12.

The `_passenger` field is set to a null reference, which will be used to flag a free agent. The `idNum()` method simply returns the id assigned to the agent when the object is created while the `isFree()` method examines the `_passenger` field to

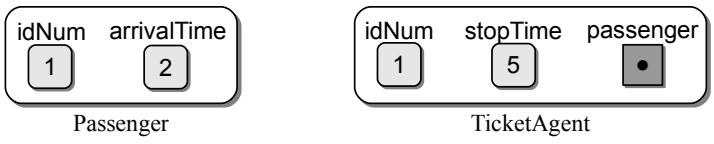


Figure 8.12: Sample `Passenger` and `TicketAgent` objects.

determine if the agent is free. The `isFinished()` method is used to determine if the passenger currently being served by this agent has completed her transaction. This method only flags the transaction as having been completed. To actually end the transaction, `stopService()` must be called. `stopService()` sets the `_passenger` field to `None` to again indicate the agent is free and returns the passenger object. To begin a transaction, `startService()` is called, which assigns the appropriate fields with the supplied arguments.

Listing 8.7 The `TicketAgent` class defined in the `simpeople.py` module.

```
# Used to store and manage information related to an airline ticket agent.
class TicketAgent :
   # Creates a ticket agent object.
  def __init__( self, idNum ):
    self._idNum = idNum
    self._passenger = None
    self._stopTime = -1

   # Gets the ticket agent's id number.
  def idNum( self ):
    return self._idNum

   # Determines if the ticket agent is free to assist a passenger.
  def isFree( self ):
    return self._passenger is None

   # Determines if the ticket agent has finished helping the passenger.
  def isFinished( self, curTime ):
    return self._passenger is not None and self._stopTime == curTime

   # Indicates the ticket agent has begun assisting a passenger.
  def startService( self, passenger, stopTime ):
    self._passenger = passenger
    self._stopTime = stopTime

   # Indicates the ticket agent has finished helping the passenger.
  def stopService( self ):
    thePassenger = self._passenger
    self._passenger = None
    return thePassenger
```

Simulation Class

Finally, we construct the `TicketCounterSimulation` class, which is provided in Listing 8.8, to manage the actual simulation. This class will contain the various components, methods, and data values required to perform a discrete event simulation. A sample instance is illustrated in Figure 8.13.

The first step in the constructor is to initialize three simulation parameters. Note the `_arriveProb` is the probability of a passenger arriving during the current time step using the formula described earlier. A queue is created, which will be

Listing 8.8 The `simulation.py` module.

```
# Implementation of the main simulation class.
from array import Array
from llistqueue import Queue
from people import TicketAgent, Passenger

class TicketCounterSimulation :
   # Create a simulation object.
  def __init__( self, numAgents, numMinutes, betweenTime, serviceTime ):
     # Parameters supplied by the user.
    self._arriveProb = 1.0 / betweenTime
    self._serviceTime = serviceTime
    self._numMinutes = numMinutes

     # Simulation components.
    self._passengerQ = Queue()
    self._theAgents = Array( numAgents )
    for i in range( numAgents ) :
      self._theAgents[i] = TicketAgent(i+1)

     # Computed during the simulation.
    self._totalWaitTime = 0
    self._numPassengers = 0

   # Run the simulation using the parameters supplied earlier.
  def run( self ):
    for curTime in range(self._numMinutes + 1) :
      self._handleArrival( curTime )
      self._handleBeginService( curTime )
      self._handleEndService( curTime )

   # Print the simulation results.
  def printResults( self ):
    numServed = self._numPassengers - len(self._passengerQ)
    avgWait = float( self._totalWaitTime ) / numServed
    print( "" )
    print( "Number of passengers served = ", numServed )
    print( "Number of passengers remaining in line = %d" %
           len(self._passengerQ) )
    print( "The average wait time was %4.2f minutes." % avgWait )

  # The remaining methods that have yet to be implemented.
  # def _handleArrive( curTime ):        # Handles simulation rule #1.
  # def _handleBeginService( curTime ):  # Handles simulation rule #2.
  # def _handleEndService( curTime ):    # Handles simulation rule #3.
```

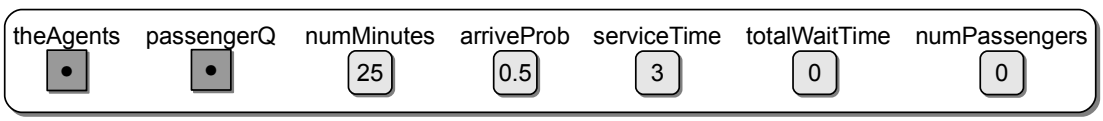


Figure 8.13: A sample `TicketCounterSimulation` object.

used to represent the line in which passengers must wait until they are served by a ticket agent. The ticket agents are represented as an array of `Agent` objects. The individual objects are instantiated and each is assigned an id number, starting with 1. Two data fields are needed to store data collected during the actual simulation. The first is the summation of the time each passenger has to wait in line before being served, and the second keeps track of the number of passengers in the simulation. The latter will also be used to assign an id to each passenger.

The simulation is performed by calling the `run()` method, which simulates the ticking of the clock by performing a count-controlled loop keeping track of the current time in `curTime`. The loop executes until `_numMinutes` have elapsed. The events of the simulation are also performed during the terminating minute, hence, the need for the `_numMinutes + 1` in the `range()` function. During each iteration of the loop, the three simulation rules outlined earlier are handled by the respective `_handleXYZ()` helper methods. The `_handleArrival()` method determines if a passenger arrives during the current time step and handles that arrival. `_handleBeginService()` determines if any agents are free and allows the next passenger(s) in line to begin their transaction. The `_handleEndService()` determines which of the current transactions have completed, if any, and flags a passenger departure. The implementation of the helper methods is left as an exercise.

After running the simulation, the `printResults()` method is called to print the results. When the simulation terminates there may be some passengers remaining in the queue who have not yet been assisted. Thus, we need to determine how many passengers have exited the queue, which indicates the number of passenger wait times included in the `_totalWaitTime` field. The average wait time is simply the total wait time divided by the number of passengers served.

The last component of our program is the driver module, which is left as an exercise. The driver extracts the simulation parameters from the user and then creates and uses a `TicketCounterSimulation` object to perform the simulation. To produce the same results shown earlier, you will need to seed the random number generator with the value 4500 before running the simulation:

```
random.seed( 4500 )
```

In a typical experiment, a simulation is performed multiple times varying the parameters with each execution. Table 8.1 illustrates the results of a single experiment with multiple executions of the simulation. Note the significant change in the average wait time when increasing the number of ticket agents by one in the last two sets of experiments.

Exercises

8.1 Determine the worst case time-complexity for each operation defined in the `TicketCounterSimulation` class.

Num Minutes	Num Agents	Average Service	Time Between	Average Wait	Passengers Served	Passengers Remaining
100	2	3	2	2.49	49	2
500	2	3	2	3.91	240	0
1000	2	3	2	10.93	490	14
5000	2	3	2	15.75	2459	6
10000	2	3	2	21.17	4930	18
100	2	4	2	10.60	40	11
500	2	4	2	49.99	200	40
1000	2	4	2	95.72	400	104
5000	2	4	2	475.91	2000	465
10000	2	4	2	949.61	4000	948
100	3	4	2	0.51	51	0
500	3	4	2	0.50	240	0
1000	3	4	2	1.06	501	3
5000	3	4	2	1.14	2465	0
10000	3	4	2	1.21	4948	0

Table 8.1: Sample results of the ticket counter simulation experiment.

8.2 Hand execute the following code and show the contents of the resulting queue:

```
values = Queue()
for i in range( 16 ) :
  if i % 3 == 0 :
    values.enqueue( i )
```

8.3 Hand execute the following code and show the contents of the resulting queue:

```
values = Queue()
for i in range( 16 ) :
  if i % 3 == 0 :
    values.enqueue( i )
  elif i % 4 == 0 :
    values.dequeue()
```

8.4 Implement the remaining methods of the `TicketCounterSimulation` class.

8.5 Modify the `TicketCounterSimulation` class to use seconds for the time units instead of minutes. Run an experiment with multiple simulations and produce a table like Table 8.1.

8.6 Design and implement a function that reverses the order of the items in a queue. Your solution may only use the operations defined by the Queue ADT, but you are free to use other data structures if necessary.

Programming Projects

8.1 Implement the Priority Queue ADT using each of the following:

(a) sorted Python list (b) sorted linked list (c) unsorted linked list

8.2 A ***deque*** (pronounced "deck") is similar to a queue, except that elements can be enqueued at either end and dequeued from either end. Define a Deque ADT and then provide an implementation for your definition.

8.3 Design and implement a ToDo List ADT in which each entry can be assigned a priority and the entries with the highest priority are performed first.

8.4 Printers can be connected to the network and made available to many users. To manage multiple print jobs and to provide fair access, networked printers use print queues. Design, implement, and test a computer program to simulate a print queue that evaluates the average wait time.

8.5 Modify your simulation from Programming Project 8.4 to use a priority queue for each print job. The priorities should range from 0 to 20 with 0 being the highest priority. Use a random number generator to determine the priority of each job.

8.6 Design, implement, and test a computer program to simulate a telephone customer service center. Your simulation should evaluate the average time customers have to wait on hold.

8.7 Design, implement, and test a computer program to simulate a bank. Your simulation should evaluate the average time customers have to wait in line before they are served by a teller.

8.8 Redesign the `TicketCounterClass` to implement a generic simulation class from which a user can derive their own simulation classes.

8.9 Design, implement, and test a computer program to simulate the checkout at a grocery store. Your simulation must allow for multiple checkout lines and allow customers at the end of a line to move to another checkout line. This simulation differs from the one described in the chapter. For this simulation, you will have to accommodate the multiple checkout lines, decide how a customer chooses a line, and decide if and when a customer moves to a new checkout line.