6.1 Stacks

A ***stack*** is a collection of objects that are inserted and removed according to the ***last-in, first-out*** (***LIFO***) principle. A user may insert objects into a stack at any time, but may only access or remove the most recently inserted object that remains (at the so-called "top" of the stack). The name "stack" is derived from the metaphor of a stack of plates in a spring-loaded, cafeteria plate dispenser. In this case, the fundamental operations involve the "pushing" and "popping" of plates on the stack. When we need a new plate from the dispenser, we "pop" the top plate off the stack, and when we add a plate, we "push" it down on the stack to become the new top plate. Perhaps an even more amusing example is a PEZ® candy dispenser, which stores mint candies in a spring-loaded container that "pops" out the topmost candy in the stack when the top of the dispenser is lifted (see Figure 6.1). Stacks are a fundamental data structure. They are used in many applications, including the following.

Example 6.1: *Internet Web browsers store the addresses of recently visited sites in a stack. Each time a user visits a new site, that site's address is "pushed" onto the stack of addresses. The browser then allows the user to "pop" back to previously visited sites using the "back" button.*

Example 6.2: *Text editors usually provide an "undo" mechanism that cancels recent editing operations and reverts to former states of a document. This undo operation can be accomplished by keeping text changes in a stack.*

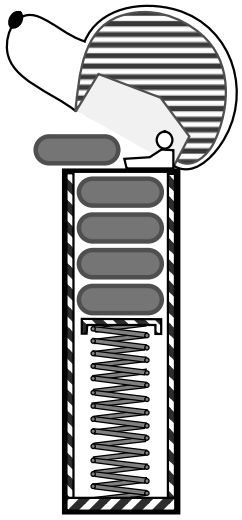

Figure 6.1: A schematic drawing of a PEZ® dispenser; a physical implementation of the stack ADT. (PEZ® is a registered trademark of PEZ Candy, Inc.)

6.1.1 The Stack Abstract Data Type

Stacks are the simplest of all data structures, yet they are also among the most important. They are used in a host of different applications, and as a tool for many more sophisticated data structures and algorithms. Formally, a stack is an abstract data type (ADT) such that an instance S supports the following two methods:

S.push(e): Add element e to the top of stack S.

S.pop( ): Remove and return the top element from the stack S; an error occurs if the stack is empty.

Additionally, let us define the following accessor methods for convenience:

S.top( ): Return a reference to the top element of stack S, without removing it; an error occurs if the stack is empty.

S.is_empty( ): Return True if stack S does not contain any elements.

len(S): Return the number of elements in stack S; in Python, we implement this with the special method __len__.

By convention, we assume that a newly created stack is empty, and that there is no a priori bound on the capacity of the stack. Elements added to the stack can have arbitrary type.

Example 6.3: *The following table shows a series of stack operations and their effects on an initially empty stack S of integers.*

Operation	Return Value	Stack Contents
S.push(5)	–	[5]
S.push(3)	–	[5, 3]
len(S)	2	[5, 3]
S.pop( )	3	[5]
S.is_empty( )	False	[5]
S.pop( )	5	[ ]
S.is_empty( )	True	[ ]
S.pop( )	"error"	[ ]
S.push(7)	–	[7]
S.push(9)	–	[7, 9]
S.top( )	9	[7, 9]
S.push(4)	–	[7, 9, 4]
len(S)	3	[7, 9, 4]
S.pop( )	4	[7, 9]
S.push(6)	–	[7, 9, 6]
S.push(8)	–	[7, 9, 6, 8]
S.pop( )	8	[7, 9, 6]

6.1.2 Simple Array-Based Stack Implementation

We can implement a stack quite easily by storing its elements in a Python list. The list class already supports adding an element to the end with the append method, and removing the last element with the pop method, so it is natural to align the top of the stack at the end of the list, as shown in Figure 6.2.

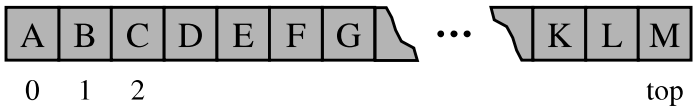

Figure 6.2: Implementing a stack with a Python list, storing the top element in the rightmost cell.

Although a programmer could directly use the list class in place of a formal stack class, lists also include behaviors (e.g., adding or removing elements from arbitrary positions) that would break the abstraction that the stack ADT represents. Also, the terminology used by the list class does not precisely align with traditional nomenclature for a stack ADT, in particular the distinction between append and push. Instead, we demonstrate how to use a list for internal storage while providing a public interface consistent with a stack.

The Adapter Pattern

The ***adapter*** design pattern applies to any context where we effectively want to modify an existing class so that its methods match those of a related, but different, class or interface. One general way to apply the adapter pattern is to define a new class in such a way that it contains an instance of the existing class as a hidden field, and then to implement each method of the new class using methods of this hidden instance variable. By applying the adapter pattern in this way, we have created a new class that performs some of the same functions as an existing class, but repackaged in a more convenient way. In the context of the stack ADT, we can adapt Python's list class using the correspondences shown in Table 6.1.

Stack Method	*Realization with Python list*
S.push(e)	L.append(e)
S.pop( )	L.pop( )
S.top( )	L[−1]
S.is_empty( )	len(L) == 0
len(S)	len(L)

Table 6.1: Realization of a stack S as an adaptation of a Python list L.

Implementing a Stack Using a Python List

We use the adapter design pattern to define an ArrayStack class that uses an underlying Python list for storage. (We choose the name ArrayStack to emphasize that the underlying storage is inherently array based.) One question that remains is what our code should do if a user calls pop or top when the stack is empty. Our ADT suggests that an error occurs, but we must decide what type of error. When pop is called on an empty Python list, it formally raises an IndexError, as lists are index-based sequences. That choice does not seem appropriate for a stack, since there is no assumption of indices. Instead, we can define a new exception class that is more appropriate. Code Fragment 6.1 defines such an Empty class as a trivial subclass of the Python Exception class.

```
class Empty(Exception):
  """Error attempting to access an element from an empty container."""
  pass
```

Code Fragment 6.1: Definition for an Empty exception class.

The formal definition for our ArrayStack class is given in Code Fragment 6.2. The constructor establishes the member self._data as an initially empty Python list, for internal storage. The rest of the public stack behaviors are implemented, using the corresponding adaptation that was outlined in Table 6.1.

Example Usage

Below, we present an example of the use of our ArrayStack class, mirroring the operations at the beginning of Example 6.3 on page 230.

```
S = ArrayStack( )                 # contents: [ ]
S.push(5)                         # contents: [5]
S.push(3)                         # contents: [5, 3]
print(len(S))                     # contents: [5, 3];       outputs 2
print(S.pop( ))                   # contents: [5];          outputs 3
print(S.is_empty( ))              # contents: [5];          outputs False
print(S.pop( ))                   # contents: [ ];          outputs 5
print(S.is_empty( ))              # contents: [ ];          outputs True
S.push(7)                         # contents: [7]
S.push(9)                         # contents: [7, 9]
print(S.top( ))                   # contents: [7, 9];       outputs 9
S.push(4)                         # contents: [7, 9, 4]
print(len(S))                     # contents: [7, 9, 4];    outputs 3
print(S.pop( ))                   # contents: [7, 9];       outputs 4
S.push(6)                         # contents: [7, 9, 6]
```

```
1  class ArrayStack:
2    """LIFO Stack implementation using a Python list as underlying storage."""
3
4    def __init__(self):
5      """Create an empty stack."""
6      self._data = [ ]                       # nonpublic list instance
7
8    def __len__(self):
9      """Return the number of elements in the stack."""
10     return len(self._data)
11
12   def is_empty(self):
13     """Return True if the stack is empty."""
14     return len(self._data) == 0
15
16   def push(self, e):
17     """Add element e to the top of the stack."""
18     self._data.append(e)                   # new item stored at end of list
19
20   def top(self):
21     """Return (but do not remove) the element at the top of the stack.
22
23     Raise Empty exception if the stack is empty.
24     """
25     if self.is_empty( ):
26       raise Empty('Stack is empty')
27     return self._data[-1]                  # the last item in the list
28
29   def pop(self):
30     """Remove and return the element from the top of the stack (i.e., LIFO).
31
32     Raise Empty exception if the stack is empty.
33     """
34     if self.is_empty( ):
35       raise Empty('Stack is empty')
36     return self._data.pop( )               # remove last item from list
```

Code Fragment 6.2: Implementing a stack using a Python list as storage.

Analyzing the Array-Based Stack Implementation

Table 6.2 shows the running times for our ArrayStack methods. The analysis directly mirrors the analysis of the list class given in Section 5.3. The implementations for top, is_empty, and len use constant time in the worst case. The $O(1)$ time for push and pop are ***amortized*** bounds (see Section 5.3.2); a typical call to either of these methods uses constant time, but there is occasionally an $O(n)$-time worst case, where n is the current number of elements in the stack, when an operation causes the list to resize its internal array. The space usage for a stack is $O(n)$.

Operation	Running Time
S.push(e)	$O(1)^*$
S.pop( )	$O(1)^*$
S.top( )	$O(1)$
S.is_empty( )	$O(1)$
len(S)	$O(1)$

*amortized

Table 6.2: Performance of our array-based stack implementation. The bounds for push and pop are amortized due to similar bounds for the list class. The space usage is $O(n)$, where n is the current number of elements in the stack.

Avoiding Amortization by Reserving Capacity

In some contexts, there may be additional knowledge that suggests a maximum size that a stack will reach. Our implementation of ArrayStack from Code Fragment 6.2 begins with an empty list and expands as needed. In the analysis of lists from Section 5.4.1, we emphasized that it is more efficient in practice to construct a list with initial length n than it is to start with an empty list and append n items (even though both approaches run in $O(n)$ time).

As an alternate model for a stack, we might wish for the constructor to accept a parameter specifying the maximum capacity of a stack and to initialize the _data member to a list of that length. Implementing such a model requires significant changes relative to Code Fragment 6.2. The size of the stack would no longer be synonymous with the length of the list, and pushes and pops of the stack would not require changing the length of the list. Instead, we suggest maintaining a separate integer as an instance variable that denotes the current number of elements in the stack. Details of such an implementation are left as Exercise C-6.17.

6.1.3 Reversing Data Using a Stack

As a consequence of the LIFO protocol, a stack can be used as a general tool to reverse a data sequence. For example, if the values 1, 2, and 3 are pushed onto a stack in that order, they will be popped from the stack in the order 3, 2, and then 1.

This idea can be applied in a variety of settings. For example, we might wish to print lines of a file in reverse order in order to display a data set in decreasing order rather than increasing order. This can be accomplished by reading each line and pushing it onto a stack, and then writing the lines in the order they are popped. An implementation of such a process is given in Code Fragment 6.3.

```
1  def reverse_file(filename):
2    """Overwrite given file with its contents line-by-line reversed."""
3    S = ArrayStack( )
4    original = open(filename)
5    for line in original:
6      S.push(line.rstrip('\n'))        # we will re-insert newlines when writing
7    original.close( )
8
9    # now we overwrite with contents in LIFO order
10   output = open(filename, 'w')     # reopening file overwrites original
11   while not S.is_empty( ):
12     output.write(S.pop( ) + '\n')  # re-insert newline characters
13   output.close( )
```

Code Fragment 6.3: A function that reverses the order of lines in a file.

One technical detail worth noting is that we intentionally strip trailing newlines from lines as they are read, and then re-insert newlines after each line when writing the resulting file. Our reason for doing this is to handle a special case in which the original file does not have a trailing newline for the final line. If we exactly echoed the lines read from the file in reverse order, then the original last line would be followed (without newline) by the original second-to-last line. In our implementation, we ensure that there will be a separating newline in the result.

The idea of using a stack to reverse a data set can be applied to other types of sequences. For example, Exercise R-6.5 explores the use of a stack to provide yet another solution for reversing the contents of a Python list (a recursive solution for this goal was discussed in Section 4.4.1). A more challenging task is to reverse the order in which elements are stored within a stack. If we were to move them from one stack to another, they would be reversed, but if we were to then replace them into the original stack, they would be reversed again, thereby reverting to their original order. Exercise C-6.18 explores a solution for this task.

6.1.4 Matching Parentheses and HTML Tags

In this subsection, we explore two related applications of stacks, both of which involve testing for pairs of matching delimiters. In our first application, we consider arithmetic expressions that may contain various pairs of grouping symbols, such as

- Parentheses: "(" and ")"
- Braces: "{" and "}"
- Brackets: "[" and "]"

Each opening symbol must match its corresponding closing symbol. For example, a left bracket, "[," must match a corresponding right bracket, "]," as in the expression [(5+x)-(y+z)]. The following examples further illustrate this concept:

- Correct: ()(()){([()])}
- Correct: ((()(()){([()])}))
- Incorrect:)(()){([()])}
- Incorrect: ({[])}
- Incorrect: (

We leave the precise definition of a matching group of symbols to Exercise R-6.6.

An Algorithm for Matching Delimiters

An important task when processing arithmetic expressions is to make sure their delimiting symbols match up correctly. Code Fragment 6.4 presents a Python implementation of such an algorithm. A discussion of the code follows.

```
1  def is_matched(expr):
2    """Return True if all delimiters are properly match; False otherwise."""
3    lefty = '({['                                  # opening delimiters
4    righty = ')}]'                                 # respective closing delims
5    S = ArrayStack()
6    for c in expr:
7      if c in lefty:
8        S.push(c)                                  # push left delimiter on stack
9      elif c in righty:
10       if S.is_empty():
11         return False                             # nothing to match with
12       if righty.index(c) != lefty.index(S.pop()):
13         return False                             # mismatched
14   return S.is_empty( )                           # were all symbols matched?
```

Code Fragment 6.4: Function for matching delimiters in an arithmetic expression.

We assume the input is a sequence of characters, such as `'[(5+x)-(y+z)]'`. We perform a left-to-right scan of the original sequence, using a stack S to facilitate the matching of grouping symbols. Each time we encounter an opening symbol, we push that symbol onto S, and each time we encounter a closing symbol, we pop a symbol from the stack S (assuming S is not empty), and check that these two symbols form a valid pair. If we reach the end of the expression and the stack is empty, then the original expression was properly matched. Otherwise, there must be an opening delimiter on the stack without a matching symbol.

If the length of the original expression is n, the algorithm will make at most n calls to push and n calls to pop. Those calls run in a total of $O(n)$ time, even considering the amortized nature of the $O(1)$ time bound for those methods. Given that our selection of possible delimiters, ({[, has constant size, auxiliary tests such as c in lefty and righty.index(c) each run in $O(1)$ time. Combining these operations, the matching algorithm on a sequence of length n runs in $O(n)$ time.

Matching Tags in a Markup Language

Another application of matching delimiters is in the validation of markup languages such as HTML or XML. HTML is the standard format for hyperlinked documents on the Internet and XML is an extensible markup language used for a variety of structured data sets. We show a sample HTML document and a possible rendering in Figure 6.3.

```
<body>
<center>
<h1> The Little Boat </h1>
</center>
<p> The storm tossed the little
boat like a cheap sneaker in an
old washing machine.  The three
drunken fishermen were used to
such treatment, of course, but
not the tree salesman, who even as
a stowaway now felt that he
had overpaid for the voyage. </p>
<ol>
<li> Will the salesman die? </li>
<li> What color is the boat? </li>
<li> And what about Naomi? </li>
</ol>
</body>
```

(a)

The Little Boat

The storm tossed the little boat like a cheap sneaker in an old washing machine. The three drunken fishermen were used to such treatment, of course, but not the tree salesman, who even as a stowaway now felt that he had overpaid for the voyage.

1. Will the salesman die?
2. What color is the boat?
3. And what about Naomi?

(b)

Figure 6.3: Illustrating HTML tags. (a) An HTML document; (b) its rendering.

In an HTML document, portions of text are delimited by ***HTML tags***. A simple opening HTML tag has the form "<name>" and the corresponding closing tag has the form "</name>". For example, we see the <body> tag on the first line of Figure 6.3(a), and the matching </body> tag at the close of that document. Other commonly used HTML tags that are used in this example include:

- body: document body
- h1: section header
- center: center justify
- p: paragraph
- ol: numbered (ordered) list
- li: list item

Ideally, an HTML document should have matching tags, although most browsers tolerate a certain number of mismatching tags. In Code Fragment 6.5, we give a Python function that matches tags in a string representing an HTML document. We make a left-to-right pass through the raw string, using index j to track our progress and the find method of the str class to locate the '<' and '>' characters that define the tags. Opening tags are pushed onto the stack, and matched against closing tags as they are popped from the stack, just as we did when matching delimiters in Code Fragment 6.4. By similar analysis, this algorithm runs in $O(n)$ time, where n is the number of characters in the raw HTML source.

```
1  def is_matched_html(raw):
2    """Return True if all HTML tags are properly match; False otherwise."""
3    S = ArrayStack( )
4    j = raw.find('<')                      # find first '<' character (if any)
5    while j != −1:
6      k = raw.find('>', j+1)               # find next '>' character
7      if k == −1:
8        return False                       # invalid tag
9      tag = raw[j+1:k]                     # strip away < >
10     if not tag.startswith('/'):          # this is opening tag
11       S.push(tag)
12     else:                                # this is closing tag
13       if S.is_empty( ):
14         return False                     # nothing to match with
15       if tag[1:] != S.pop( ):
16         return False                     # mismatched delimiter
17     j = raw.find('<', k+1)               # find next '<' character (if any)
18   return S.is_empty( )                   # were all opening tags matched?
```

Code Fragment 6.5: Function for testing if an HTML document has matching tags.

6.2 Queues

Another fundamental data structure is the ***queue***. It is a close "cousin" of the stack, as a queue is a collection of objects that are inserted and removed according to the ***first-in, first-out*** (***FIFO***) principle. That is, elements can be inserted at any time, but only the element that has been in the queue the longest can be next removed.

We usually say that elements enter a queue at the back and are removed from the front. A metaphor for this terminology is a line of people waiting to get on an amusement park ride. People waiting for such a ride enter at the back of the line and get on the ride from the front of the line. There are many other applications of queues (see Figure 6.4). Stores, theaters, reservation centers, and other similar services typically process customer requests according to the FIFO principle. A queue would therefore be a logical choice for a data structure to handle calls to a customer service center, or a wait-list at a restaurant. FIFO queues are also used by many computing devices, such as a networked printer, or a Web server responding to requests.

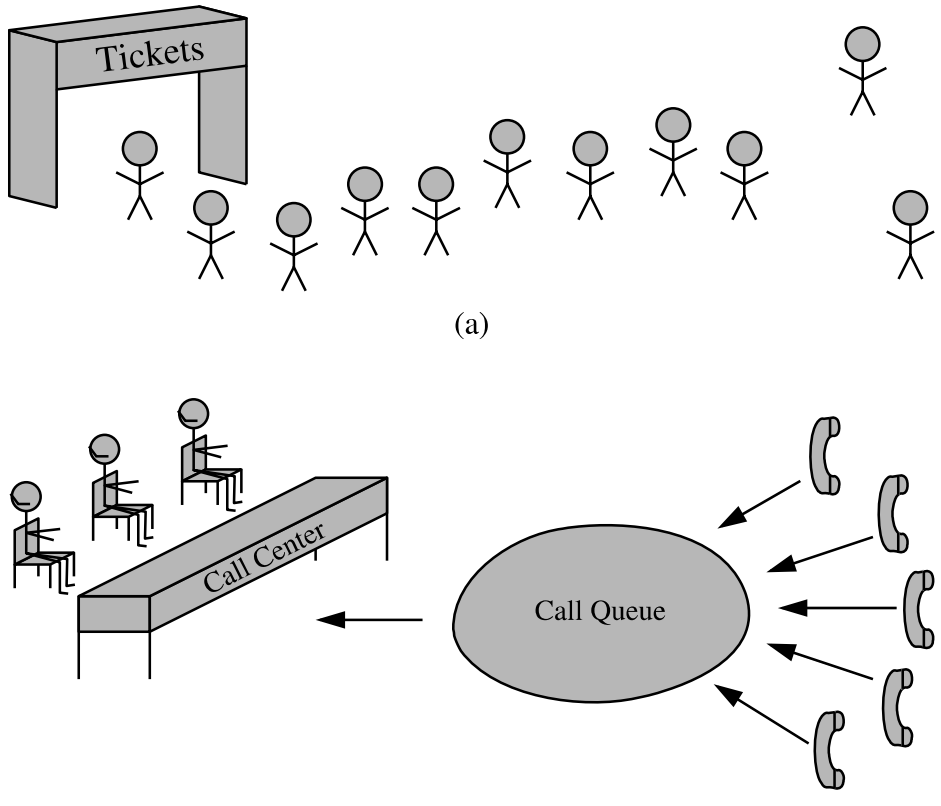

Figure 6.4: Real-world examples of a first-in, first-out queue. (a) People waiting in line to purchase tickets; (b) phone calls being routed to a customer service center.

6.2.1 The Queue Abstract Data Type

Formally, the queue abstract data type defines a collection that keeps objects in a sequence, where element access and deletion are restricted to the ***first*** element in the queue, and element insertion is restricted to the back of the sequence. This restriction enforces the rule that items are inserted and deleted in a queue according to the first-in, first-out (FIFO) principle. The ***queue*** abstract data type (ADT) supports the following two fundamental methods for a queue Q:

Q.enqueue(e): Add element e to the back of queue Q.

Q.dequeue( ): Remove and return the first element from queue Q; an error occurs if the queue is empty.

The queue ADT also includes the following supporting methods (with first being analogous to the stack's top method):

Q.first( ): Return a reference to the element at the front of queue Q, without removing it; an error occurs if the queue is empty.

Q.is_empty( ): Return True if queue Q does not contain any elements.

len(Q): Return the number of elements in queue Q; in Python, we implement this with the special method __len__.

By convention, we assume that a newly created queue is empty, and that there is no a priori bound on the capacity of the queue. Elements added to the queue can have arbitrary type.

Example 6.4: *The following table shows a series of queue operations and their effects on an initially empty queue Q of integers.*

Operation	Return Value	first $\leftarrow Q \leftarrow$ last
Q.enqueue(5)	–	[5]
Q.enqueue(3)	–	[5, 3]
len(Q)	2	[5, 3]
Q.dequeue( )	5	[3]
Q.is_empty( )	False	[3]
Q.dequeue( )	3	[ ]
Q.is_empty( )	True	[ ]
Q.dequeue( )	"error"	[ ]
Q.enqueue(7)	–	[7]
Q.enqueue(9)	–	[7, 9]
Q.first( )	7	[7, 9]
Q.enqueue(4)	–	[7, 9, 4]
len(Q)	3	[7, 9, 4]
Q.dequeue( )	7	[9, 4]

6.2.2 Array-Based Queue Implementation

For the stack ADT, we created a very simple adapter class that used a Python list as the underlying storage. It may be very tempting to use a similar approach for supporting the queue ADT. We could enqueue element e by calling append(e) to add it to the end of the list. We could use the syntax pop(0), as opposed to pop( ), to intentionally remove the ***first*** element from the list when dequeuing.

As easy as this would be to implement, it is tragically inefficient. As we discussed in Section 5.4.1, when pop is called on a list with a non-default index, a loop is executed to shift all elements beyond the specified index to the left, so as to fill the hole in the sequence caused by the pop. Therefore, a call to pop(0) always causes the worst-case behavior of $\Theta(n)$ time.

We can improve on the above strategy by avoiding the call to pop(0) entirely. We can replace the dequeued entry in the array with a reference to None, and maintain an explicit variable f to store the index of the element that is currently at the front of the queue. Such an algorithm for dequeue would run in $O(1)$ time. After several dequeue operations, this approach might lead to the configuration portrayed in Figure 6.5.

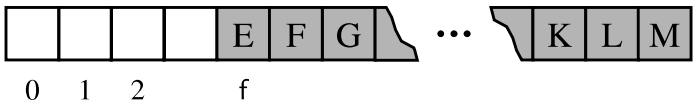

Figure 6.5: Allowing the front of the queue to drift away from index 0.

Unfortunately, there remains a drawback to the revised approach. In the case of a stack, the length of the list was precisely equal to the size of the stack (even if the underlying array for the list was slightly larger). With the queue design that we are considering, the situation is worse. We can build a queue that has relatively few elements, yet which are stored in an arbitrarily large list. This occurs, for example, if we repeatedly enqueue a new element and then dequeue another (allowing the front to drift rightward). Over time, the size of the underlying list would grow to $O(m)$ where m is the ***total*** number of enqueue operations since the creation of the queue, rather than the current number of elements in the queue.

This design would have detrimental consequences in applications in which queues have relatively modest size, but which are used for long periods of time. For example, the wait-list for a restaurant might never have more than 30 entries at one time, but over the course of a day (or a week), the overall number of entries would be significantly larger.

Using an Array Circularly

In developing a more robust queue implementation, we allow the front of the queue to drift rightward, and we allow the contents of the queue to "wrap around" the end of an underlying array. We assume that our underlying array has fixed length N that is greater that the actual number of elements in the queue. New elements are enqueued toward the "end" of the current queue, progressing from the front to index $N-1$ and continuing at index 0, then 1. Figure 6.6 illustrates such a queue with first element E and last element M.

Figure 6.6: Modeling a queue with a circular array that wraps around the end.

Implementing this circular view is not difficult. When we dequeue an element and want to "advance" the front index, we use the arithmetic `f = (f + 1) % N`. Recall that the % operator in Python denotes the ***modulo*** operator, which is computed by taking the remainder after an integral division. For example, 14 divided by 3 has a quotient of 4 with remainder 2, that is, $\frac{14}{3} = 4\frac{2}{3}$. So in Python, `14 // 3` evaluates to the quotient 4, while `14 % 3` evaluates to the remainder 2. The modulo operator is ideal for treating an array circularly. As a concrete example, if we have a list of length 10, and a front index 7, we can advance the front by formally computing `(7+1) % 10`, which is simply 8, as 8 divided by 10 is 0 with a remainder of 8. Similarly, advancing index 8 results in index 9. But when we advance from index 9 (the last one in the array), we compute `(9+1) % 10`, which evaluates to index 0 (as 10 divided by 10 has a remainder of zero).

A Python Queue Implementation

A complete implementation of a queue ADT using a Python list in circular fashion is presented in Code Fragments 6.6 and 6.7. Internally, the queue class maintains the following three instance variables:

_data: is a reference to a list instance with a fixed capacity.

_size: is an integer representing the current number of elements stored in the queue (as opposed to the length of the _data list).

_front: is an integer that represents the index within _data of the first element of the queue (assuming the queue is not empty).

We initially reserve a list of moderate size for storing data, although the queue formally has size zero. As a technicality, we initialize the _front index to zero.

When front or dequeue are called with no elements in the queue, we raise an instance of the Empty exception, defined in Code Fragment 6.1 for our stack.

```
1  class ArrayQueue:
2    """FIFO queue implementation using a Python list as underlying storage."""
3    DEFAULT_CAPACITY = 10          # moderate capacity for all new queues
4
5    def __init__(self):
6      """Create an empty queue."""
7      self._data = [None] * ArrayQueue.DEFAULT_CAPACITY
8      self._size = 0
9      self._front = 0
10
11   def __len__(self):
12     """Return the number of elements in the queue."""
13     return self._size
14
15   def is_empty(self):
16     """Return True if the queue is empty."""
17     return self._size == 0
18
19   def first(self):
20     """Return (but do not remove) the element at the front of the queue.
21
22     Raise Empty exception if the queue is empty.
23     """
24     if self.is_empty():
25       raise Empty('Queue is empty')
26     return self._data[self._front]
27
28   def dequeue(self):
29     """Remove and return the first element of the queue (i.e., FIFO).
30
31     Raise Empty exception if the queue is empty.
32     """
33     if self.is_empty():
34       raise Empty('Queue is empty')
35     answer = self._data[self._front]
36     self._data[self._front] = None                    # help garbage collection
37     self._front = (self._front + 1) % len(self._data)
38     self._size -= 1
39     return answer
```

Code Fragment 6.6: Array-based implementation of a queue (continued in Code Fragment 6.7).

```
40   def enqueue(self, e):
41     """Add an element to the back of queue."""
42     if self._size == len(self._data):
43       self._resize(2 * len(self.data))          # double the array size
44     avail = (self._front + self._size) % len(self._data)
45     self._data[avail] = e
46     self._size += 1
47
48   def _resize(self, cap):                       # we assume cap >= len(self)
49     """Resize to a new list of capacity >= len(self)."""
50     old = self._data                            # keep track of existing list
51     self._data = [None] * cap                   # allocate list with new capacity
52     walk = self._front
53     for k in range(self._size):                 # only consider existing elements
54       self._data[k] = old[walk]                 # intentionally shift indices
55       walk = (1 + walk) % len(old)              # use old size as modulus
56     self._front = 0                             # front has been realigned
```

Code Fragment 6.7: Array-based implementation of a queue (continued from Code Fragment 6.6).

The implementation of __len__ and is_empty are trivial, given knowledge of the size. The implementation of the front method is also simple, as the _front index tells us precisely where the desired element is located within the _data list, assuming that list is not empty.

Adding and Removing Elements

The goal of the enqueue method is to add a new element to the back of the queue. We need to determine the proper index at which to place the new element. Although we do not explicitly maintain an instance variable for the back of the queue, we compute the location of the next opening based on the formula:

```
avail = (self._front + self._size) % len(self._data)
```

Note that we are using the size of the queue as it exists ***prior*** to the addition of the new element. For example, consider a queue with capacity 10, current size 3, and first element at index 5. The three elements of such a queue are stored at indices 5, 6, and 7. The new element should be placed at index $(\text{front} + \text{size}) = 8$. In a case with wrap-around, the use of the modular arithmetic achieves the desired circular semantics. For example, if our hypothetical queue had 3 elements with the first at index 8, our computation of (8+3) % 10 evaluates to 1, which is perfect since the three existing elements occupy indices 8, 9, and 0.

When the dequeue method is called, the current value of self._front designates the index of the value that is to be removed and returned. We keep a local reference to the element that will be returned, setting answer = self._data[self._front] just prior to removing the reference to that object from the list, with the assignment self._data[self._front] = None. Our reason for the assignment to None relates to Python's mechanism for reclaiming unused space. Internally, Python maintains a count of the number of references that exist to each object. If that count reaches zero, the object is effectively inaccessible, thus the system may reclaim that memory for future use. (For more details, see Section 15.1.2.) Since we are no longer responsible for storing a dequeued element, we remove the reference to it from our list so as to reduce that element's reference count.

The second significant responsibility of the dequeue method is to update the value of _front to reflect the removal of the element, and the presumed promotion of the second element to become the new first. In most cases, we simply want to increment the index by one, but because of the possibility of a wrap-around configuration, we rely on modular arithmetic as originally described on page 242.

Resizing the Queue

When enqueue is called at a time when the size of the queue equals the size of the underlying list, we rely on a standard technique of doubling the storage capacity of the underlying list. In this way, our approach is similar to the one used when we implemented a DynamicArray in Section 5.3.1.

However, more care is needed in the queue's _resize utility than was needed in the corresponding method of the DynamicArray class. After creating a temporary reference to the old list of values, we allocate a new list that is twice the size and copy references from the old list to the new list. While transferring the contents, we intentionally realign the front of the queue with index 0 in the new array, as shown in Figure 6.7. This realignment is not purely cosmetic. Since the modular arithmetic depends on the size of the array, our state would be flawed had we transferred each element to its same index in the new array.

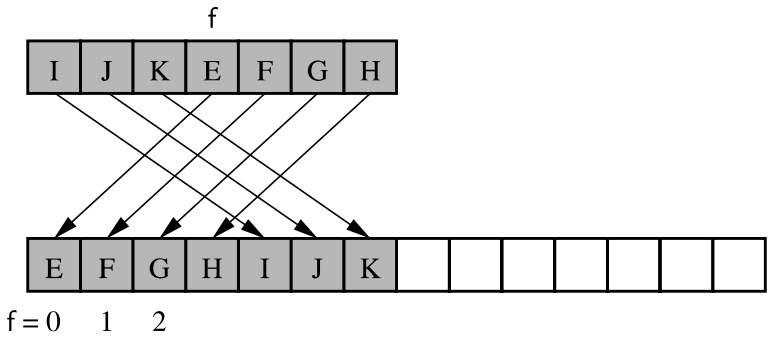

Figure 6.7: Resizing the queue, while realigning the front element with index 0.

Shrinking the Underlying Array

A desirable property of a queue implementation is to have its space usage be $\Theta(n)$ where n is the current number of elements in the queue. Our ArrayQueue implementation, as given in Code Fragments 6.6 and 6.7, does not have this property. It expands the underlying array when enqueue is called with the queue at full capacity, but the dequeue implementation never shrinks the underlying array. As a consequence, the capacity of the underlying array is proportional to the maximum number of elements that have ever been stored in the queue, not the current number of elements.

We discussed this very issue on page 200, in the context of dynamic arrays, and in subsequent Exercises C-5.16 through C-5.20 of that chapter. A robust approach is to reduce the array to half of its current size, whenever the number of elements stored in it falls below *one fourth* of its capacity. We can implement this strategy by adding the following two lines of code in our dequeue method, just after reducing self._size at line 38 of Code Fragment 6.6, to reflect the loss of an element.

```
if 0 < self._size < len(self._data) // 4:
  self._resize(len(self._data) // 2)
```

Analyzing the Array-Based Queue Implementation

Table 6.3 describes the performance of our array-based implementation of the queue ADT, assuming the improvement described above for occasionally shrinking the size of the array. With the exception of the _resize utility, all of the methods rely on a constant number of statements involving arithmetic operations, comparisons, and assignments. Therefore, each method runs in worst-case $O(1)$ time, except for enqueue and dequeue, which have ***amortized*** bounds of $O(1)$ time, for reasons similar to those given in Section 5.3.

Operation	Running Time
Q.enqueue(e)	$O(1)^*$
Q.dequeue( )	$O(1)^*$
Q.first( )	$O(1)$
Q.is_empty( )	$O(1)$
len(Q)	$O(1)$

*amortized

Table 6.3: Performance of an array-based implementation of a queue. The bounds for enqueue and dequeue are amortized due to the resizing of the array. The space usage is $O(n)$, where n is the current number of elements in the queue.