# Data Structures in Python

## 7. Queues and Deques

**Prof. Moheb Ramzy Girgis**
**Department of Computer Science**
**Faculty of Science**
**Minia University**

## Queues

- Another fundamental data structure is the ***queue***.
- *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.
- 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.

2

## 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):



## The Queue Abstract Data Type

  - ***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.
  - ***Q.len()*****:** Return the number of elements in queue Q.
- 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 1:** The following table shows a series of queue operations and their effects on an initially empty queue Q of integers.

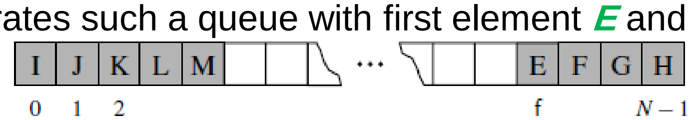
# The Queue Abstract Data Type

| Operation | Return Value | first ← $Q$ ← last |
|---|---|---|
| Q.enqueue(5) | – | [5] |
| Q.enqueue(3) | – | [5, 3] |
| **Q.len()** | 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] |
| **Q.len()** | 3 | [7, 9, 4] |
| Q.dequeue( ) | 7 | [9, 4] |



# Array-Based Queue Implementation

- In our queue implementation, we use a ***circular array***. We
  - maintain an explicit variable ***f*** to store the index of the element that is currently at the ***front*** of the queue,
  - allow the front of the queue to drift rightward, and
  - 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 than 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***.
- The figure illustrates such a queue with first element ***E*** and last element ***M***.



# Array-Based Queue Implementation

- 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.
- For example, if we have a list of length 10, and a front index 7,
  - we can advance the front by computing (7+1) % 10, which is simply 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.

❖ **A Python Queue Implementation**

- In the implementation of the queue ADT, we use a Python list in circular fashion.
- Internally, the queue class maintains the following three instance variables:



# Array-Based Queue Implementation

  - ***_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.
- We initialize the ***front*** index to zero.
- When ***first*** or ***dequeue*** are called with no elements in the queue, we raise an instance of the ***Empty exception***, defined before for our stack.
- The complete implementation of the queue ADT is presented below:

## Array-Based Queue Implementation

```
class ArrayQueue:
    """FIFO queue implementation using a Python list as underlying
    storage."""
    DEFAULT_CAPACITY = 10 # moderate capacity for all new queues
    def __init__(self):
        """Create an empty queue."""
        self._data = [None] * ArrayQueue.DEFAULT_CAPACITY
        self._size = 0
        self._front = 0
    def len(self):
        """Return the number of elements in the queue."""
        return self._size
    def is_empty(self):
        """Return True if the queue is empty."""
        return self._size == 0
    def first(self):
        """Return (but do not remove) the element at the front of the queue.
        Raise Empty exception if the queue is empty."""
        if self.is_empty( ):
            raise Empty('Queue is empty')
        return self._data[self._front]
```

9

## Array-Based Queue Implementation

```
    def dequeue(self):
        """Remove and return the first element of the queue (i.e., FIFO).
        Raise Empty exception if the queue is empty."""
        if self.is_empty( ):
            raise Empty('Queue is empty')
        answer = self._data[self._front]
        self._data[self._front] = None # help garbage collection
        self._front = (self._front + 1) % len(self._data)
        self._size -= 1
        return answer
    def enqueue(self, e):
        """Add an element to the back of queue."""
        if self._size == len(self._data):
            self.resize(2 * len(self._data)) # double the array size
        avail = (self._front + self._size) % len(self._data)
        self._data[avail] = e
        self._size += 1
```

Data Structures in Python - Prof. Moheb Ramzy Girgis Dept. of Computer Science - Faculty of Science Minia University

10

# Array-Based Queue Implementation

```
def resize(self, cap):  # we assume cap >= len(self)
    """Resize to a new list of capacity >= len(self)."""
    old = self._data # keep track of existing list
    self._data = [None] * cap # allocate list with new capacity
    walk = self._front
    for k in range(self._size): # only consider existing elements
        self._data[k] = old[walk] # intentionally shift indices
        walk = (1 + walk) % len(old) # use old size as modulus
    self._front = 0 # front has been realigned
```

- The implementation of ***len*** and ***is_empty*** are trivial, given knowledge of the ***size***.
- The implementation of the ***first*** 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.



# Array-Based Queue Implementation

> ***Adding Elements***

- The goal of the ***enqueue*** method is to add a new element to the back of the queue.
- We determine the proper index at which to place the new element 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 (front+size) = 8.

## Array-Based Queue Implementation

- 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.

➢ ***Removing Elements***

- 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.



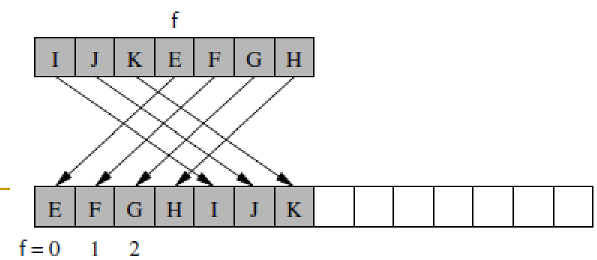
## Array-Based Queue Implementation

- The reason for the assignment to ***None*** is to remove the reference to dequeued element from our list so that the system can reclaim that unused memory space for future use (garbage collection).
- The second significant responsibility of the ***dequeue*** method is to update the value of ***_front*** to reflect the removal of the element, and the 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 use the assignment:

  self._front = (self._front + 1) % len(self._data)
- Finally, the ***dequeue*** method decreases the size by 1.

# Array-Based Queue Implementation

- ***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*.
  - Firstly, we create a temporary reference to the old list of values, then 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 the figure:






# Array-Based Queue Implementation

- **Example 2**: In this example, we use our ***ArrayQueue*** class, to do the operations of Example 1 on Slide 5:

```
Q = ArrayQueue()
Q.enqueue(5)           # contents: [5]
Q.enqueue(3)           # contents: [5, 3]
print(Q.len())         # contents: [5, 3];      outputs 2
print(Q.dequeue())     # contents: [3];         outputs 5
print(Q.is_empty())    # contents: [3];         outputs False
print(Q.dequeue())     # contents: [ ];         outputs 3
print(Q.is_empty())    # contents: [ ];         outputs True
Q.enqueue(7)           # contents: [7]
Q.enqueue(9)           # contents: [7, 9]
print(Q.first())       # contents: [7, 9];      outputs 7
Q.enqueue(4)           # contents: [7, 9, 4]
print(Q.len())         # contents: [7, 9, 4];   outputs 3
print(Q.dequeue())     # contents: [9, 4];      outputs 7
```

## Problem: Identifying Palindromes

- To demonstrate the use of stacks and queues, we look at a simple problem: *identifying palindromes*.
- A ***palindrome*** is a string that reads the same forwards as backwards.
- Some famous palindromes are:
  - "A man, a plan, a canal - Panama!"
  - "Won ton? Not now!"
  - "Madam, I'm Adam."
  - "Eve."
- As you can see, the rules for what is a palindrome are somewhat lenient. Typically, we do not worry about punctuation, spaces, or matching the case of letters.
- An input file holds a separate string on each line. An output file is created, repeating each of the input lines and stating whether or not it is a palindrome.



## Problem: Identifying Palindromes

- Finally, summary statistics are written to a Message Dialog Box.
- The program reads a line of input and creates a new stack and a new queue, and it repeatedly pushes each letter from the input line onto the stack, and also enqueues it onto the queue. To simplify comparison later, the actual characters pushed and enqueued are the lowercase versions of the characters in the string.
- When all of the characters of the line have been processed, the program repeatedly pops a letter from the stack and dequeues a letter from the queue. As long as these letters match each other for the entire way through this process, we have a palindrome, because we are comparing the letters from the forward view of the string (from the queue) to the letters from the backward view of the string (from the stack).
- Now we are ready to write the main algorithm:

# Problem: Identifying Palindromes

- *Main Algorithm*

Initialize expression counts
While there are still lines to process
  Read first input line
  Increment the total number of strings
  Echo print the current string to the output file
  Process the current string
  if (!stillPalindrome)
    Increment the count of non palindromes
    Write "Not a palindrome" to the output file
  else
    Increment the count of palindromes
    Write "Is a palindrome" to the output file
End While
Write summary information to the output message dialog box

- The summary information includes:
  - Total Number Of Strings
  - Number Of Palindromes
  - Number Of Non Palindromes

- The details of how it is determined (whether or not the current string is a palindrome) are hidden in the phrase "*Process the current string*". The description of that algorithm is as follows:

19

# Problem: Identifying Palindromes

- *Process the current string*

Create a new stack
Create a new queue
For each character in the string
  if the character is a letter
    Change the character to lowercase
    Push the character onto the stack
    enqueue the character onto the queue
End For
Set stillPalindrome to true
While (there are still more characters in the structures && the string can still be a palindrome)
  Pop character1 from the stack
  dequeue a character2 from the queue
  if (character1 != character2)
    Set stillPalindrome to false
End While

- The implementation of the algorithm for identifying palindromes is left as an exercise.

# Problem: Identifying Palindromes

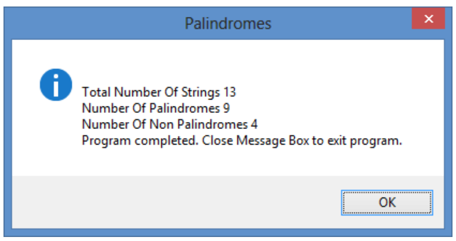
**Input File**

```
A man, a plan, a canal, Panama
amanaplanacanalpanama
This is not a palindrone!
aaaaaaaaaaa
a
aaaaaaaaaaaabaaaaaaaaaaaaaaaaaaaaaaaaaa
aAaaaaaaaabaaaaaaAaaa
This string is too long xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
bob
dan
Madam, I'm Adam.
eve
Eve
```

**Output File**

```
String  1: A man, a plan, a canal, Panama
        Is a palindrome.
String  2: amanaplanacanalpanama
        Is a palindrome.
String  3: This is not a palindrone!
        Not a palindrome
String  4: aaaaaaaaaaa
        Is a palindrome.
String  5: a
        Is a palindrome.
String  6: aaaaaaaaaaaabaaaaaaaaaaaaaaaaaaaaaaaaaa
        Not a palindrome
String  7: aAaaaaaaaabaaaaaaAaaa
        Is a palindrome.
String  8: This string is too long xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
        Not a palindrome
String  9: bob
        Is a palindrome.
String 10: dan
        Not a palindrome
String 11: Madam, I'm Adam.
        Is a palindrome.
String 12: eve
        Is a palindrome.
String 13: Eve
        Is a palindrome.
```

**The summary information**






# Double-Ended Queues

- Now, we consider a queue-like data structure that supports insertion and deletion at both the front and the back of the queue.
- Such a structure is called a ***double ended queue***, or ***deque***, which is usually pronounced “deck”.
- The ***deque ADT*** is more general than both the ***stack*** and the ***queue*** ADTs.
- The extra generality can be useful in some applications.
- For example, consider a restaurant using a queue to maintain a waitlist.
  - Occasionally, the first person might be removed from the queue only to find that a table was not available; typically, the restaurant will re-insert the person at the *first* position in the queue.
  - It may also be that a customer at the end of the queue may grow impatient and leave the restaurant.

## The Deque Abstract Data Type

- To provide a symmetrical abstraction, the ***deque ADT*** is defined so that ***deque D*** supports the following methods:
  - ***D.add_first(e)*****:** Add element e to the front of ***deque D***.
  - ***D.add_last(e)*****:** Add element e to the back of ***deque D***.
  - ***D.delete_first( )*****:** Remove and return the first element from ***deque D***; an error occurs if the ***deque*** is empty.
  - ***D.delete_last( )*****:** Remove and return the last element from deque D; an error occurs if the ***deque*** is empty.
- Additionally, the ***deque ADT*** will include the following accessors:
  - ***D.first()*****:** Return (but do not remove) the first element of deque D; an error occurs if the ***deque*** is empty.
  - ***D.last()*****:** Return (but do not remove) the last element of deque D; an error occurs if the ***deque*** is empty.
  - ***D.is_empty( )*****:** Return True if ***deque D*** does not contain any elements.
  - ***D.len()*****:** Return the number of elements in ***deque D***.

23

## The Deque Abstract Data Type

- **Example 3:** The following table shows a series of operations and their effects on an initially empty ***deque D*** of integers.

| Operation | Return Value | Deque |
|---|---|---|
| D.add_last(5) | – | [5] |
| D.add_first(3) | – | [3, 5] |
| D.add_first(7) | – | [7, 3, 5] |
| D.first( ) | 7 | [7, 3, 5] |
| D.delete_last( ) | 5 | [7, 3] |
| D.len() | 2 | [7, 3] |
| D.delete_last( ) | 3 | [7] |
| D.delete_last( ) | 7 | [ ] |
| D.add_first(6) | – | [6] |
| D.last( ) | 6 | [6] |
| D.add_first(8) | – | [8, 6] |
| D.is_empty( ) | False | [8, 6] |
| D.last( ) | 6 | [8, 6] |

## Implementing a Deque with a Circular Array

- We can implement the ***deque ADT*** in much the same way as the ***ArrayQueue*** class.
- So, the details of an ***ArrayDeque*** implementation is left as an exercise.
- We recommend maintaining the same three instance variables: *data*, *size*, and *front*.
- Whenever we need to know the index of the back of the ***deque***, or the first available slot beyond the back of the ***deque***, we use modular arithmetic for the computation.
- For example, our implementation of the last() method uses the index

  back = (self._front + self._size − 1) % len(self._data)
- Our implementation of the ArrayDeque.add_lastmethod is essentially the same as that for ArrayQueue.enqueue, including the reliance on a resize utility.



## Implementing a Deque with a Circular Array

- Likewise, the implementation of the ***ArrayDeque.delete_first*** method is the same as ***ArrayQueue.dequeue***.
- Implementations of ***add_first*** and ***delete_last*** use similar techniques.
- One subtlety is that a call to ***add_first*** may need to wrap around the beginning of the array, so we rely on modular arithmetic to circularly *decrement* the index, as

self._front = (self._front − 1) % len(self._data) # cyclic shift

## Deques in the Python Collections Module

- An implementation of a ***deque*** class is available in Python's standard ***collections*** module.
- A summary of the most commonly used behaviors of the ***collections.deque*** class is given in the table below.

| Our Deque ADT | collections.deque | Description |
|---|---|---|
| **D.len()** | len(D) | number of elements |
| D.add_first() | D.appendleft() | add to beginning |
| D.add_last() | D.append() | add to end |
| D.delete_first() | D.popleft() | remove from beginning |
| D.delete_last() | D.pop() | remove from end |
| D.first() | D[0] | access first element |
| D.last() | D[−1] | access last element |
| | D[j] | access arbitrary entry by index |
| | D[j] = val | modify arbitrary entry by index |
| | D.clear() | clear all contents |
| | D.rotate(k) | circularly shift rightward k steps |
| | D.remove(e) | remove first matching element |
| | D.count(e) | count number of matches for e |



## Deques in the Python Collections Module

- The ***collections.deque*** interface was chosen to be consistent with the naming conventions of Python's list class, for which:
  - ***append*** and ***pop*** act at the end of the list.
  - ***appendleft*** and ***popleft*** act at the beginning of the list.
- The library ***deque*** also mimics a ***list*** in that it is an indexed sequence, allowing arbitrary access or modification using the ***D[j]*** syntax.
- The library ***deque constructor*** also supports an optional ***maxlen*** parameter to force a fixed-length deque.
- However, if a call to ***append*** at either end is invoked when the ***deque*** is full, it does not throw an error; instead, it causes one element to be dropped from the opposite side.
- That is, calling ***appendleft*** when the ***deque*** is full causes an implicit pop from the right side to make room for the new element.

## Deques in the Python Collections Module

- **Example** Run

```
import collections
d = collections.deque('abcdefg')
print('Deque:', d)
print('Length:', len(d))
print('Left end:', d[0])
print('Right end:', d[-1])
d.remove('c')
print('remove(c):', d)
d.append('h') # Add to the right
print('append    :', d)
d.appendleft('s') # Add to the left
print('appendleft:', d)
d = collections.deque('abc', 5) #maxlen=5
print('Deque:', d)
d.append('h')
print('append    :', d)
d.append('k')
print('append    :', d)
d.append('m')
print('append    :', d)
```

```
d = collections.deque(range(5))
print('Normal:', d)
d.rotate(2) # Rotate right 2 steps
print('Right rotation:', d)
d = collections.deque(range(5))
print('Normal:', d)
d.rotate(-2) # Rotate left 2 steps
print('Left rotation :', d)
```

```
Deque: deque(['a', 'b', 'c', 'd', 'e', 'f', 'g'])
Length: 7
Left end: a
Right end: g
remove(c): deque(['a', 'b', 'd', 'e', 'f', 'g'])
append    : deque(['a', 'b', 'd', 'e', 'f', 'g', 'h'])
appendleft: deque(['s', 'a', 'b', 'd', 'e', 'f', 'g', 'h'])
Deque: deque(['a', 'b', 'c'], maxlen=5)
append    : deque(['a', 'b', 'c', 'h'], maxlen=5)
append    : deque(['a', 'b', 'c', 'h', 'k'], maxlen=5)
append    : deque(['b', 'c', 'h', 'k', 'm'], maxlen=5)
Normal: deque([0, 1, 2, 3, 4])
Right rotation: deque([3, 4, 0, 1, 2])
Normal: deque([0, 1, 2, 3, 4])
Left rotation : deque([2, 3, 4, 0, 1])
```

29

## Priority Queues

- An ordinary queue is a ***first-in, first-out*** data structure. Elements are appended to the end of the queue and removed from the beginning.
- In a ***priority queue***, elements are assigned with priorities. When accessing elements, the element with the highest priority is removed first. A priority queue has a ***highest-in, first-out*** behavior. For example, the emergency room in a hospital assigns priority numbers to patients; the patient with the highest priority is treated first.
- A ***priority queue*** can be implemented using a ***heap***.
- In Python, the ***heap data structure*** is available using the ***Heap queue (or heapq) module***.
- The ***heapq module*** provides the heap data structure that is mainly used to represent a ***priority queue***. The property of this data structure in Python is that each time the smallest heap element is popped (min-heap).

30

## Priority Queues

- Whenever elements are pushed or popped, heap structure is maintained.
- The ***heap[0]*** element returns the smallest element each time.

➢ ***Some operations on the heap***

- ***heapify(iterable)***: is used to convert ***iterable*** into a heap.
- ***heappush(heap, e)***: is used to insert ***e*** into a heap. The order is adjusted, so that the heap structure is maintained.
- ***heappop(heap)***: is used to remove and return the smallest element from the heap. The order is adjusted, so that the heap structure is maintained.
- **Example:**

```
import heapq    # importing "heapq" to implement heap queue
lst = [5, 7, 9, 1, 3]         # initializing list
heapq.heapify(lst)       # using heapify to convert list into heap
# printing created heap
print ("The created heap is : ", end="")
print (list(lst))
```



## Priority Queues

```
# using heappush() to push elements into heap
heapq.heappush(lst, 4)             # pushes 4
# printing modified heap
print ("The modified heap after push is : ", end="")
print (list(lst))
# using heappop() to pop smallest element
print ("The popped and smallest element is : ", end="")
print (heapq.heappop(lst))
```

Run

```
The created heap is : [1, 3, 9, 7, 5]
The modified heap after push is : [1, 3, 4, 7, 5, 9]
The popped and smallest element is : 1
```

➢ ***Priority Queue Implementation Using Python heap***

- The implementation of a priority queue class, named ***MyPriorityQueue***, using Python ***Heap queue*** , is given in the following listing:

## Priority Queues

```
# Importing "heapq" to implement priority queue
import heapq
class MyPriorityQueue:
   def __init__(self):
      """Create an empty priority queue."""
      self._data = []
      heapq.heapify(self._data)
   def enqueue(self, e):
      """Add an element e to the priority queue."""
      heapq.heappush(self._data, e)
   def dequeue(self):
      """Remove and return the smallest element in the queue.
      Raise Empty exception if the queue is empty.""“
      if self.is_empty( ):
         raise Empty('Queue is empty')
      return heapq.heappop(self._data)
   def len(self):
      """Return the number of elements in the queue."""
      return len(self._data)
```

33

## Priority Queues

```
    def first(self):
      """Return (but do not remove) the smallest element in the queue.
      Raise Empty exception if the queue is empty."""
      if self.is_empty( ):
         raise Empty('Queue is empty')
      return self._data[0]
   def is_empty(self):
      """Return True if the queue is empty."""
      return len(self._data) == 0
```

- **Example:**

```
# Create an empty priority queue
pq = MyPriorityQueue()
print("Is queue empty?", pq.is_empty())
# Enqueue some elements into the created priority queue
pq.enqueue(5)
pq.enqueue(7)
pq.enqueue(9)
pq.enqueue(1)
pq.enqueue(3)
```

34

# Priority Queues

```
print("No. of elements in the priority queue:", pq.len())
print("The smallest element in the priority queue:", pq.first())
# Dequeue all elements from the created priority queue and display them
print("The contents of the priority queue:")
while not pq.is_empty():
   print(pq.dequeue(), end=" ")
print()
print("Is queue empty?", pq.is_empty())
```

Run

```
Is queue empty? True
No. of elements in the queue after enqueuing some elements: 5
The smallest element in the priority queue: 1
The contents of the priority queue:
1 3 5 7 9
Is queue empty? True
```