
Figure 3.12: A Queue of Python Data Objects)

## 3.5 Queues

We now turn our attention to another linear data structure. This one is called **queue**. Like stacks, queues are relatively simple and yet can be used to solve a wide range of important problems.

### 3.5.1 What Is a Queue?

A queue is an ordered collection of items where the addition of new items happens at one end, called the "rear," and the removal of existing items occurs at the other end, commonly called the "front." As an element enters the queue it starts at the rear and makes its way toward the front, waiting until that time when it is the next element to be removed.

The most recently added item in the queue must wait at the end of the collection. The item that has been in the collection the longest is at the front. This ordering principle is sometimes called FIFO, first-in first-out. It is also known as "first-come first-served."

The simplest example of a queue is the typical line that we all participate in from time to time. We wait in a line for a movie, we wait in the check-out line at a grocery store, and we wait in the cafeteria line (so that we can pop the tray stack). Well-behaved lines, or queues, are very restrictive in that they have only one way in and only one way out. There is no jumping in the middle and no leaving before you have waited the necessary amount of time to get to the front. Figure 3.12 shows a simple queue of Python data objects.

Computer science also has common examples of queues. Our computer laboratory has 30 computers networked with a single printer. When students want to print, their print tasks "get in line" with all the other printing tasks that are waiting. The first task in is the next to be completed. If you are last in line, you must wait for all the other tasks to print ahead of you. We will explore this interesting example in more detail later.

In addition to printing queues, operating systems use a number of different queues to control processes within a computer. The scheduling of what gets done next is typically based on a queuing algorithm that tries to execute programs as quickly as possible and serve as many users as it can. Also, as we type, sometimes keystrokes get ahead of the characters that appear on the screen. This is due to the computer doing other work at that moment. The keystrokes are being placed in a queue-like buffer so that they can eventually be displayed on the screen in the proper order.

| Queue Operation | Queue Contents | Return Value |
|---|---|---|
| `q.is_empty()` | `[]` | `True` |
| `q.enqueue(4)` | `[4]` | |
| `q.enqueue('dog')` | `['dog',4]` | |
| `q.enqueue(True)` | `[True,'dog',4]` | |
| `q.size()` | `[True,'dog',4]` | `3` |
| `q.is_empty()` | `[True,'dog',4]` | `False` |
| `q.enqueue(8.4)` | `[8.4,True,'dog',4]` | |
| `q.dequeue()` | `[8.4,True,'dog']` | `4` |
| `q.dequeue()` | `[8.4,True]` | `'dog'` |
| `q.size()` | `[8.4,True]` | `2` |

Table 3.5: Example Queue Operations

### 3.5.2 The Queue Abstract Data Type

The queue abstract data type is defined by the following structure and operations. A queue is structured, as described above, as an ordered collection of items which are added at one end, called the "rear," and removed from the other end, called the "front." Queues maintain a FIFO ordering property. The queue operations are given below.

- Queue() creates a new queue that is empty. It needs no parameters and returns an empty queue.
- enqueue(item) adds a new item to the rear of the queue. It needs the item and returns nothing.
- dequeue() removes the front item from the queue. It needs no parameters and returns the item. The queue is modified.
- is_empty() tests to see whether the queue is empty. It needs no parameters and returns a boolean value.
- size() returns the number of items in the queue. It needs no parameters and returns an integer.

As an example, if we assume that $q$ is a queue that has been created and is currently empty, then Table 3.5 shows the results of a sequence of queue operations. The queue contents are shown such that the front is on the right. 4 was the first item enqueued so it is the first item returned by dequeue.

### 3.5.3 Implementing A Queue in Python

It is again appropriate to create a new class for the implementation of the abstract data type queue. As before, we will use the power and simplicity of the list collection to build the internal representation of the queue.

We need to decide which end of the list to use as the rear and which to use as the front. The implementation shown below assumes that the rear is at position 0 in the list. This allows us to use the insert function on lists to add new elements to the rear of the queue. The pop operation

can be used to remove the front element (the last element of the list). Recall that this also means that enqueue will be $O(n)$ and dequeue will be $O(1)$.

```
# Completed implementation of a queue ADT
class Queue:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return self.items == []

    def enqueue(self, item):
        self.items.insert(0,item)

    def dequeue(self):
        return self.items.pop()

    def size(self):
        return len(self.items)
```

### Self Check

Suppose you have the following series of queue operations:

```
q = Queue()
q.enqueue('hello')
q.enqueue('dog')
q.enqueue(3)
q.dequeue()
```

What items are left in the queue?

1. 'hello', 'dog'
2. 'dog', 3
3. 'hello', 3
4. 'hello', 'dog', 3

## 3.5.4 Simulation: Hot Potato

One of the typical applications for showing a queue in action is to simulate a real situation that requires data to be managed in a FIFO manner. To begin, let's consider the children's game Hot Potato. In this game (see Figure 3.13) children line up in a circle and pass an item from neighbour to neighbour as fast as they can. At a certain point in the game, the action is stopped and the child who has the item (the potato) is removed from the circle. Play continues until only one child is left.

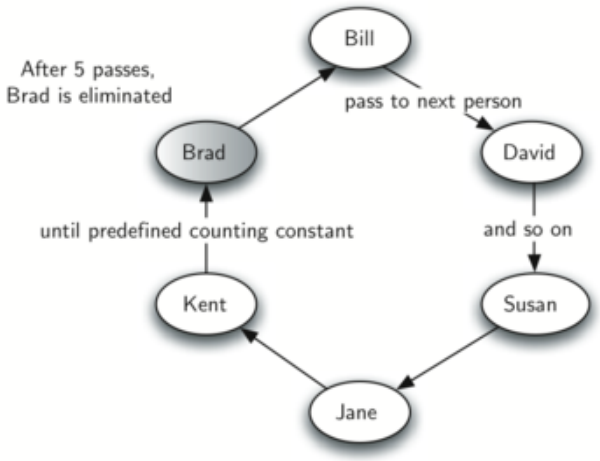


Figure 3.13: A Six Person Game of Hot Potato)

This game is a modern-day equivalent of the famous Josephus problem. Based on a legend about the famous first-century historian Flavius Josephus, the story is told that in the Jewish revolt against Rome, Josephus and 39 of his comrades held out against the Romans in a cave. With defeat imminent, they decided that they would rather die than be slaves to the Romans. They arranged themselves in a circle. One man was designated as number one, and proceeding clockwise they killed every seventh man. Josephus, according to the legend, was among other things an accomplished mathematician. He instantly figured out where he ought to sit in order to be the last to go. When the time came, instead of killing himself, he joined the Roman side. You can find many different versions of this story. Some count every third man and some allow the last man to escape on a horse. In any case, the idea is the same.

We will implement a general simulation of Hot Potato. Our program will input a list of names and a constant, call it "num" to be used for counting. It will return the name of the last person remaining after repetitive counting by num. What happens at that point is up to you.

To simulate the circle, we will use a queue (see Figure 3.14). Assume that the child holding the potato will be at the front of the queue. Upon passing the potato, the simulation will simply dequeue and then immediately enqueue that child, putting her at the end of the line. She will then wait until all the others have been at the front before it will be her turn again. After num dequeue/enqueue operations, the child at the front will be removed permanently and another cycle will begin. This process will continue until only one name remains (the size of the queue is 1).

A call to the hot_potato function using 7 as the counting constant returns Susan.

```
import Queue  # As previously defined

def hot_potato(name_list, num):
    sim_queue = Queue()
    for name in name_list:
        sim_queue.enqueue(name)
```

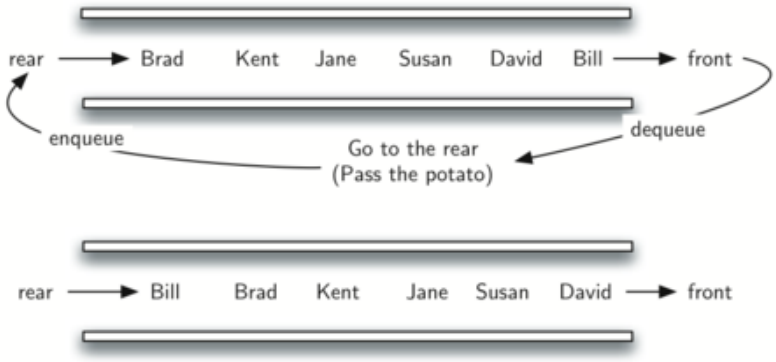


Figure 3.14: A Queue Implementation of Hot Potato)

```
    while sim_queue.size() > 1:
        for i in range(num):
            sim_queue.enqueue(sim_queue.dequeue())

        sim_queue.dequeue()

    return sim_queue.dequeue()

print(hot_potato(["Bill", "David", "Susan", "Jane", "Kent",
    "Brad"], 7))
```

Note that in this example the value of the counting constant is greater than the number of names in the list. This is not a problem since the queue acts like a circle and counting continues back at the beginning until the value is reached. Also, notice that the list is loaded into the queue such that the first name on the list will be at the front of the queue. Bill in this case is the first item in the list and therefore moves to the front of the queue. A variation of this implementation, described in the exercises, allows for a random counter.

### 3.5.5 Simulation: Printing Tasks

A more interesting simulation allows us to study the behavior of the printing queue described earlier in this section. Recall that as students send printing tasks to the shared printer, the tasks are placed in a queue to be processed in a first-come first-served manner. Many questions arise with this configuration. The most important of these might be whether the printer is capable of handling a certain amount of work. If it cannot, students will be waiting too long for printing and may miss their next class.

Consider the following situation in a computer science laboratory. On any average day about 10 students are working in the lab at any given hour. These students typically print up to twice during that time, and the length of these tasks ranges from 1 to 20 pages. The printer in the lab is older, capable of processing 10 pages per minute of draft quality. The printer could be switched to give better quality, but then it would produce only five pages per minute. The slower printing speed could make students wait too long. What page rate should be used?

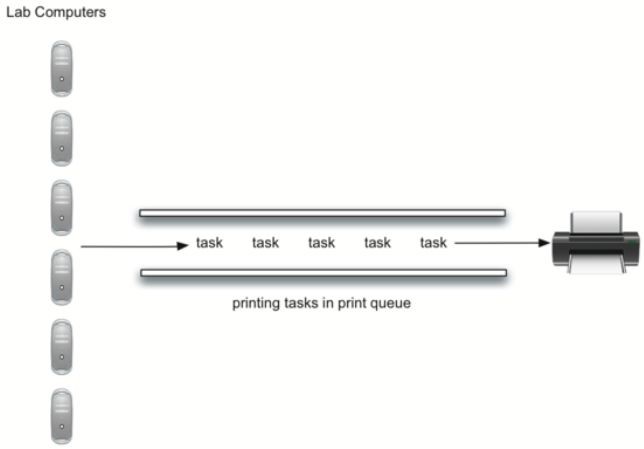


Figure 3.15: Computer Science Laboratory Printing Queue)

We could decide by building a simulation that models the laboratory. We will need to construct representations for students, printing tasks, and the printer (Figure 3.15). As students submit printing tasks, we will add them to a waiting list, a queue of print tasks attached to the printer. When the printer completes a task, it will look at the queue to see if there are any remaining tasks to process. Of interest for us is the average amount of time students will wait for their papers to be printed. This is equal to the average amount of time a task waits in the queue.

To model this situation we need to use some probabilities. For example, students may print a paper from 1 to 20 pages in length. If each length from 1 to 20 is equally likely, the actual length for a print task can be simulated by using a random number between 1 and 20 inclusive. This means that there is equal chance of any length from 1 to 20 appearing.

If there are 10 students in the lab and each prints twice, then there are 20 print tasks per hour on average. What is the chance that at any given second, a print task is going to be created? The way to answer this is to consider the ratio of tasks to time. Twenty tasks per hour means that on average there will be one task every 180 seconds:

$$\frac{20 \text{ tasks}}{1 \text{ hour}} \times \frac{1 \text{ hour}}{60 \text{ minutes}} \times \frac{1 \text{ minute}}{60 \text{ seconds}} = \frac{1 \text{ task}}{180 \text{ seconds}}$$

For every second we can simulate the chance that a print task occurs by generating a random number between 1 and 180 inclusive. If the number is 180, we say a task has been created. Note that it is possible that many tasks could be created in a row or we may wait quite a while for a task to appear. That is the nature of simulation. You want to simulate the real situation as closely as possible given that you know general parameters.

### 3.5.6 Main Simulation Steps

Here is the main simulation.

1. Create a queue of print tasks. Each task will be given a timestamp upon its arrival. The queue is empty to start.

2. For each second (current_second):
    - Does a new print task get created? If so, add it to the queue with the current_second as the timestamp.
    - If the printer is not busy and if a task is waiting,
        - Remove the next task from the print queue and assign it to the printer.
        - Subtract the timestamp from the current_second to compute the waiting time for that task.
        - Append the waiting time for that task to a list for later processing.
        - Based on the number of pages in the print task, figure out how much time will be required.
    - The printer now does one second of printing if necessary. It also subtracts one second from the time required for that task.
    - If the task has been completed, in other words the time required has reached zero, the printer is no longer busy.
3. After the simulation is complete, compute the average waiting time from the list of waiting times generated.

### 3.5.7 Python Implementation

To design this simulation we will create classes for the three real-world objects described above: Printer, Task, and PrintQueue.

The Printer class will need to track whether it has a current task. If it does, then it is busy and the amount of time needed can be computed from the number of pages in the task. The constructor will also allow the pages-per-minute setting to be initialized. The tick method decrements the internal timer and sets the printer to idle if the task is completed.

```
class Printer:
   def __init__(self, ppm):
      self.page_rate = ppm
      self.current_task = None
      self.time_remaining = 0

   def tick(self):
      if self.current_task != None:
         self.time_remaining = self.time_remaining - 1
         if self.time_remaining <= 0:
            self.current_task = None

   def busy(self):
      if self.current_task != None:
         return True
      else:
         return False
```

```
   def start_next(self,new_task):
      self.current_task = new_task
      self.time_remaining = new_task.get_pages() * 60 /
         self.page_rate
```

The Task class will represent a single printing task. When the task is created, a random number generator will provide a length from 1 to 20 pages. We have chosen to use the randrange function from the random module.

```
>>> import random
>>> random.randrange(1, 21)
18
>>> random.randrange(1, 21)
8
>>>
```

Each task will also need to keep a timestamp to be used for computing waiting time. This timestamp will represent the time that the task was created and placed in the printer queue. The wait_time method can then be used to retrieve the amount of time spent in the queue before printing begins.

```
import random

class Task:
   def __init__(self, time):
      self.timestamp = time
      self.pages = random.randrange(1, 21)

   def get_stamp(self):
      return self.timestamp

   def get_pages(self):
      return self.pages

   def wait_time(self, current_time):
      return current_time - self.timestamp
```

The main simulation implements the algorithm described above. The print_queue object is an instance of our existing queue ADT. A boolean helper function, new_print_task, decides whether a new printing task has been created. We have again chosen to use the randrange function from the random module to return a random integer between 1 and 180. Print tasks arrive once every 180 seconds. By arbitrarily choosing 180 from the range of random integers, we can simulate this random event. The simulation function allows us to set the total time and the pages per minute for the printer.

```
import Queue  # As previously defined
import Printer  # As previously defined
```

```
import Task  # As previously defined

import random

def simulation(num_seconds, pages_per_minute):

   lab_printer = Printer(pages_per_minute)
   print_queue = Queue()
   waiting_times = []

   for current_second in range(num_seconds):

     if new_print_task():
        task = Task(current_second)
        print_queue.enqueue(task)

     if (not lab_printer.busy()) and (not print_queue.is_empty()):
       next_task = print_queue.dequeue()
       waiting_times.append(next_task.wait_time(current_second))
       lab_printer.start_next(next_task)

     lab_printer.tick()

   average_wait = sum(waiting_times) / len(waiting_times)
   print("Average Wait %6.2f secs %3d tasks remaining."
      %(average_wait, print_queue.size()))

def new_print_task():
   num = random.randrange(1, 181)
   if num == 180:
      return True
   else:
      return False

for i in range(10):
   simulation(3600, 5)
```

When we run the simulation, we should not be concerned that the results are different each time. This is due to the probabilistic nature of the random numbers. We are interested in the trends that may be occurring as the parameters to the simulation are adjusted. Here are some results.

First, we will run the simulation for a period of $60$ minutes $(3,600$ seconds) using a page rate of five pages per minute. In addition, we will run $10$ independent trials. Remember that because the simulation works with random numbers each run will return different results.

```
>>>for i in range(10):
     simulation(3600, 5)

Average Wait 165.38 secs 2 tasks remaining.
Average Wait 95.07 secs 1 tasks remaining.
```

```
Average Wait 65.05 secs 2 tasks remaining.
Average Wait 99.74 secs 1 tasks remaining.
Average Wait 17.27 secs 0 tasks remaining.
Average Wait 239.61 secs 5 tasks remaining.
Average Wait 75.11 secs 1 tasks remaining.
Average Wait 48.33 secs 0 tasks remaining.
Average Wait 39.31 secs 3 tasks remaining.
Average Wait 376.05 secs 1 tasks remaining.
```

After running our 10 trials we can see that the mean average wait time is 122.155 seconds. You can also see that there is a large variation in the average weight time with a minimum average of 17.27 seconds and a maximum of 239.61 seconds. You may also notice that in only two of the cases were all the tasks completed.

Now, we will adjust the page rate to 10 pages per minute, and run the 10 trials again, with a faster page rate our hope would be that more tasks would be completed in the one hour time frame.

```
>>>for i in range(10):
     simulation(3600, 10)

Average Wait 1.29 secs 0 tasks remaining.
Average Wait 7.00 secs 0 tasks remaining.
Average Wait 28.96 secs 1 tasks remaining.
Average Wait 13.55 secs 0 tasks remaining.
Average Wait 12.67 secs 0 tasks remaining.
Average Wait 6.46 secs 0 tasks remaining.
Average Wait 22.33 secs 0 tasks remaining.
Average Wait 12.39 secs 0 tasks remaining.
Average Wait 7.27 secs 0 tasks remaining.
Average Wait 18.17 secs 0 tasks remaining.
```

The code to run the simulation is as follows:

```
import Queue  # As previously defined

import random

# Completed program for the printer simulation

class Printer:
   def __init__(self, ppm):
      self.page_rate = ppm
      self.current_task = None
      self.time_remaining = 0

   def tick(self):
      if self.current_task != None:
         self.time_remaining = self.time_remaining - 1
         if self.time_remaining <= 0:
```

```
            self.current_task = None

    def busy(self):
        if self.current_task != None:
            return True
        else:
            return False

    def start_next(self, new_task):
        self.current_task = new_task
        self.time_remaining = new_task.get_pages() * 60/self.page_rate

class Task:
    def __init__(self, time):
        self.timestamp = time
        self.pages = random.randrange(1, 21)

    def get_stamp(self):
        return self.timestamp

    def get_pages(self):
        return self.pages

    def wait_time(self, current_time):
        return current_time - self.timestamp


def simulation(num_seconds, pages_per_minute):

    lab_printer = Printer(pages_per_minute)
    print_queue = Queue()
    waiting_times = []

    for current_second in range(num_seconds):

      if new_print_task():
         task = Task(current_second)
         print_queue.enqueue(task)

      if (not lab_printer.busy()) and (not print_queue.is_empty()):
        next_task = print_queue.dequeue()
        waiting_times.append(next_task.wait_time(current_second))
        lab_printer.startNext(next_task)

      lab_printer.tick()

    average_wait = sum(waiting_times) / len(waiting_times)
    print("Average Wait %6.2f secs %3d tasks remaining."
        %(average_wait, print_queue.size()))

def new_print_task():
```

```
    num = random.randrange(1, 181)
    if num == 180:
        return True
    else:
        return False

for i in range(10):
    simulation(3600, 5)
```

### 3.5.8 Discussion

We were trying to answer a question about whether the current printer could handle the task load if it were set to print with a better quality but slower page rate. The approach we took was to write a simulation that modeled the printing tasks as random events of various lengths and arrival times.

The output above shows that with 5 pages per minute printing, the average waiting time varied from a low of 17 seconds to a high of 376 seconds (about 6 minutes). With a faster printing rate, the low value was 1 second with a high of only 28. In addition, in 8 out of 10 runs at 5 pages per minute there were print tasks still waiting in the queue at the end of the hour.

Therefore, we are perhaps persuaded that slowing the printer down to get better quality may not be a good idea. Students cannot afford to wait that long for their papers, especially when they need to be getting on to their next class. A six-minute wait would simply be too long.

This type of simulation analysis allows us to answer many questions, commonly known as "what if" questions. All we need to do is vary the parameters used by the simulation and we can simulate any number of interesting behaviors. For example,

- What if enrolment goes up and the average number of students increases by 20?
- What if it is Saturday and students are not needing to get to class? Can they afford to wait?
- What if the size of the average print task decreases since Python is such a powerful language and programs tend to be much shorter?

These questions could all be answered by modifying the above simulation. However, it is important to remember that the simulation is only as good as the assumptions that are used to build it. Real data about the number of print tasks per hour and the number of students per hour was necessary to construct a robust simulation.

#### Self Check

How would you modify the printer simulation to reflect a larger number of students? Suppose that the number of students was doubled. You make need to make some reasonable assumptions about how this simulation was put together but what would you change? Modify the code. Also suppose that the length of the average print task was cut in half. Change the code to reflect that change. Finally How would you parameterize the number of students, rather than changing the code we would like to make the number of students a parameter of the simulation.

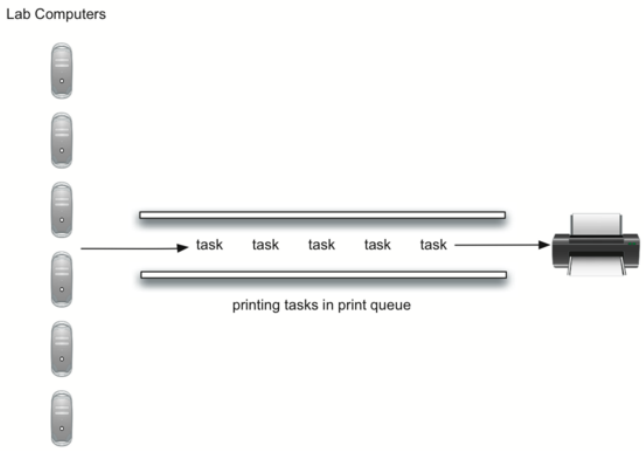


Figure 3.16: A Deque of Python Data Objects Queue)

## 3.6 Deques

We will conclude this introduction to basic data structures by looking at another variation on the theme of linear collections. However, unlike stack and queue, the deque (pronounced "deck") has very few restrictions. Also, be careful that you do not confuse the spelling of "deque" with the queue removal operation "dequeue."

### 3.6.1 What Is a Deque?

A deque, also known as a double-ended queue, is an ordered collection of items similar to the queue. It has two ends, a front and a rear, and the items remain positioned in the collection. What makes a deque different is the unrestrictive nature of adding and removing items. New items can be added at either the front or the rear. Likewise, existing items can be removed from either end. In a sense, this hybrid linear structure provides all the capabilities of stacks and queues in a single data structure. Figure 3.16 shows a deque of Python data objects.

It is important to note that even though the deque can assume many of the characteristics of stacks and queues, it does not require the LIFO and FIFO orderings that are enforced by those data structures. It is up to you to make consistent use of the addition and removal operations.

### 3.6.2 The Deque Abstract Data Type

The deque abstract data type is defined by the following structure and operations. A deque is structured, as described above, as an ordered collection of items where items are added and removed from either end, either front or rear. The deque operations are given below.

- Deque() creates a new deque that is empty. It needs no parameters and returns an empty deque.
- add_front(item) adds a new item to the front of the deque. It needs the item and returns nothing.

| Deque Operation | Deque Contents | Return value |
|---|---|---|
| d.is_empty() | [] | True |
| d.add_rear(4) | [4] | |
| d.add_rear('dog') | ['dog',4,] | |
| d.add_front('cat') | ['dog',4,'cat'] | |
| d.add_front(True) | ['dog',4,'cat',True] | |
| d.size() | ['dog',4,'cat',True] | 4 |
| d.is_empty() | ['dog',4,'cat',True] | False |
| d.add_rear(8.4) | [8.4,'dog',4,'cat',True] | |
| d.remove_rear() | ['dog',4,'cat',True] | 8.4 |
| d.remove_front() | ['dog',4,'cat'] | True |

Table 3.6: Examples of Deque Operations

- add_rear(item) adds a new item to the rear of the deque. It needs the item and returns nothing.
- remove_front() removes the front item from the deque. It needs no parameters and returns the item. The deque is modified.
- remove_rear() removes the rear item from the deque. It needs no parameters and returns the item. The deque is modified.
- is_empty() tests to see whether the deque is empty. It needs no parameters and returns a boolean value.
- size() returns the number of items in the deque. It needs no parameters and returns an integer.

As an example, if we assume that d is a deque that has been created and is currently empty, then Table 3.6 shows the results of a sequence of deque operations. Note that the contents in front are listed on the right. It is very important to keep track of the front and the rear as you move items in and out of the collection as things can get a bit confusing.

### 3.6.3 Implementing a Deque in Python

As we have done in previous sections, we will create a new class for the implementation of the abstract data type deque. Again, the Python list will provide a very nice set of methods upon which to build the details of the deque. Our implementation will assume that the rear of the deque is at position 0 in the list.

```
# Completed implementation of a deque ADT
class Deque:
   def __init__(self):
      self.items = []

   def is_empty(self):
      return self.items == []

   def add_front(self, item):
```

```
        self.items.append(item)

    def add_rear(self, item):
        self.items.insert(0,item)

    def remove_front(self):
        return self.items.pop()

    def remove_rear(self):
        return self.items.pop(0)

    def size(self):
        return len(self.items)
```

In remove_front we use the pop method to remove the last element from the list. However, in remove_rear, the pop(0) method must remove the first element of the list. Likewise, we need to use the insert method in add_rear since the append method assumes the addition of a new element to the end of the list.

You can see many similarities to Python code already described for stacks and queues. You are also likely to observe that in this implementation adding and removing items from the front is $O(1)$ whereas adding and removing from the rear is $O(n)$. This is to be expected given the common operations that appear for adding and removing items. Again, the important thing is to be certain that we know where the front and rear are assigned in the implementation.

### 3.6.4 Palindrome Checker

An interesting problem that can be easily solved using the deque data structure is the classic palindrome problem. A **palindrome** is a string that reads the same forward and backward, for example, *radar*, *toot*, and *madam*. We would like to construct an algorithm to input a string of characters and check whether it is a palindrome.

The solution to this problem will use a deque to store the characters of the string. We will process the string from left to right and add each character to the rear of the deque. At this point, the deque will be acting very much like an ordinary queue. However, we can now make use of the dual functionality of the deque. The front of the deque will hold the first character of the string and the rear of the deque will hold the last character (see Figure 3.17)

Since we can remove both of them directly, we can compare them and continue only if they match. If we can keep matching first and the last items, we will eventually either run out of characters or be left with a deque of size 1 depending on whether the length of the original string was even or odd. In either case, the string must be a palindrome.

```
import Deque  # As previously defined

def pal_checker(a_string):
    char_deque = Deque()

    for ch in a_string:
```

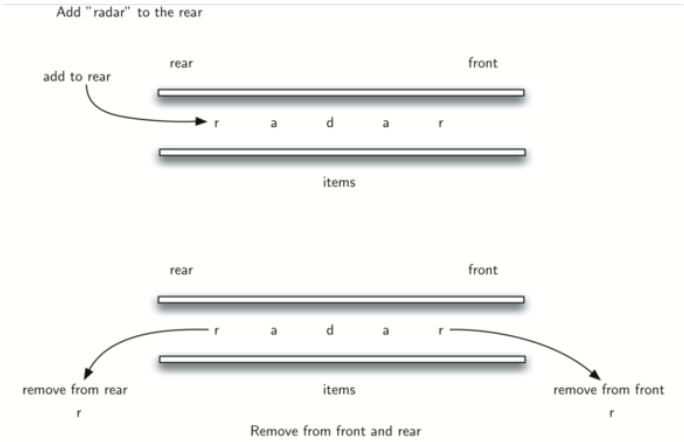


Figure 3.17: A Deque)

```
        char_deque.add_rear(ch)

    still_equal = True

    while char_deque.size() > 1 and still_equal:
        first = char_deque.remove_front()
        last = char_deque.remove_rear()
        if first != last:
            still_equal = False

    return still_equal

print(pal_checker("lsdkjfskf"))
print(pal_checker("radar"))
```

## 3.7 Lists

Throughout the discussion of basic data structures, we have used Python lists to implement the abstract data types presented. The list is a powerful, yet simple, collection mechanism that provides the programmer with a wide variety of operations. However, not all programming languages include a list collection. In these cases, the notion of a list must be implemented by the programmer.

A list is a collection of items where each item holds a relative position with respect to the others. More specifically, we will refer to this type of list as an unordered list. We can consider the list as having a first item, a second item, a third item, and so on. We can also refer to the beginning of the list (the first item) or the end of the list (the last item). For simplicity we will assume that lists cannot contain duplicate items.

For example, the collection of integers $54, 26, 93, 17, 77$, and $31$ might represent a simple unordered list of exam scores. Note that we have written them as comma-delimited values,