Python Data Structures

Outline

- Basic data structures
- Stacks
- Queques
- Desques

Basic data structures

- Linear structures
  - Stacks, queues, desques, and lists
  - Data collections where
    - Items are ordered depending on how they are added/removed
    - Position relative to the other elements that came before and after
  - Two ends of linear structures
    - Where to add and where to remove?

Stacks

Stacks

- Example:

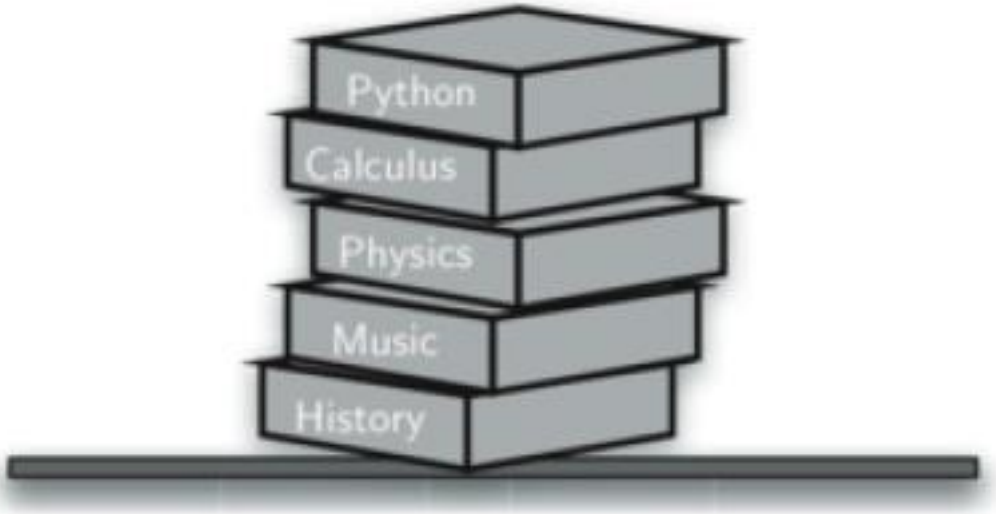


Figure 3.1: A Stack of Books

Stacks

- A stack, or “push-down stack”, is an ordered collection of items where the addition of new items and the removal of existing items always takes place at the same end.

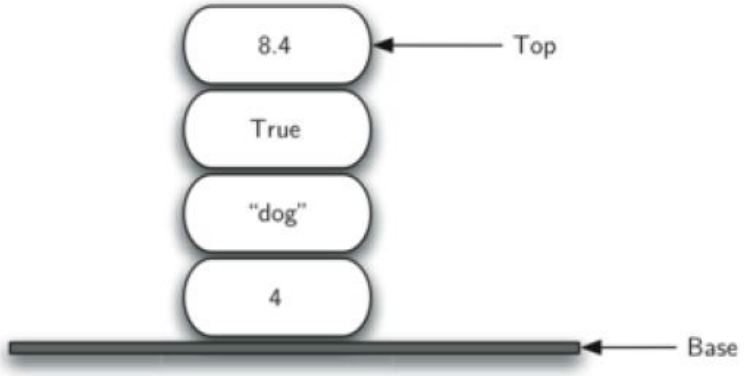


Figure 3.2: A Stack of Primitive Python Objects

Stacks

- Stacks are last-in first-out (LIFO)

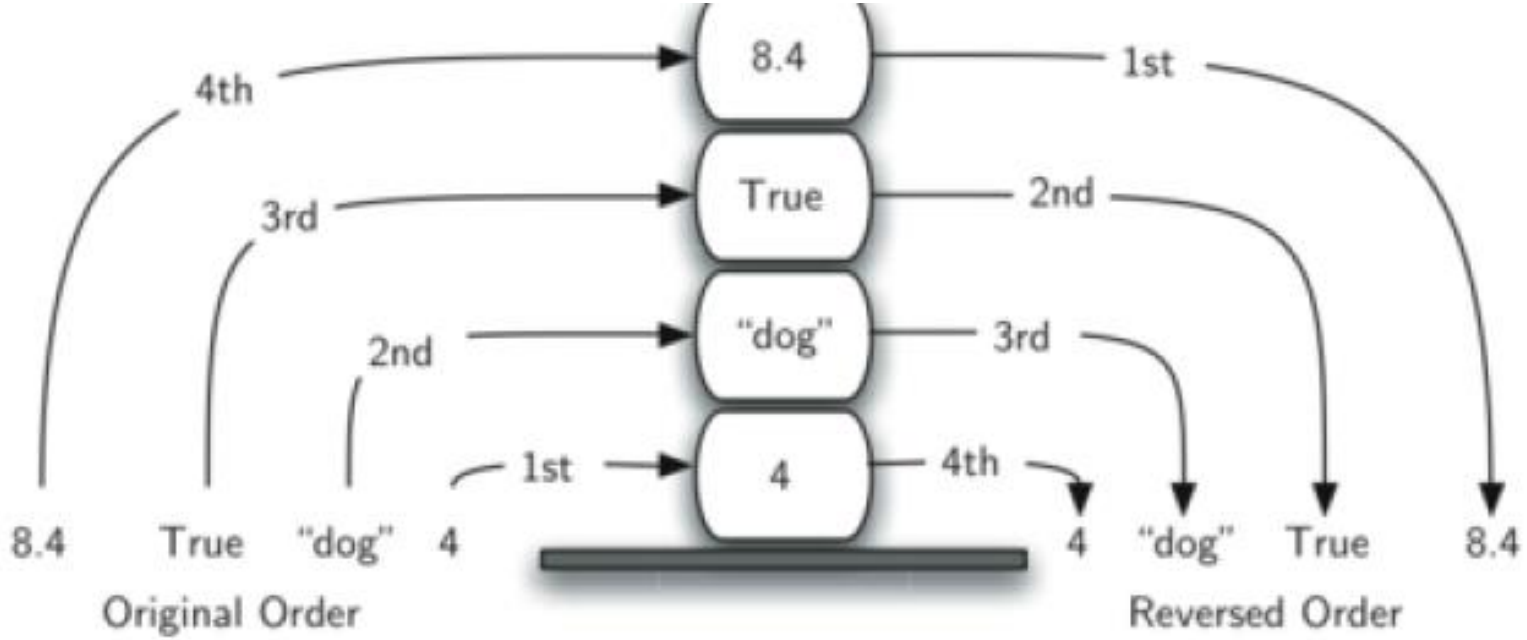


Figure 3.3: The Reversal Property of Stacks

Stacks

- Stack abstract data type:
  - Stack() creates a new stack that is empty. It needs no parameters and returns an empty stack.
  - push(item) adds a new item to the top of the stack. It needs the item and returns nothing.
  - pop() removes the top item from the stack. It needs no parameters and returns the item. The stack is modified.

Stacks

- Stack abstract data type (continued):
  - peek() returns the top item from the stack but does not remove it. It needs no parameters.The stack is not modified.
  - is_empty() tests to see whether the stack is empty. It needs no parameters and returns a boolean value.
  - size() returns the number of items on the stack. It needs no parameters and returns an integer.

Stack class implementation

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

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

    def push(self, item):
        self.items.append(item)

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

    def peek(self):
        return self.items[len(self.items)-1]

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

Stacks

- Stack operation examples:

Stack Operation	Stack Contents	Return Value
`s.is_empty()`	`[]`	`True`
`s.push(4)`	`[4]`	
`s.push('dog')`	`[4,'dog']`	
`s.peek()`	`[4,'dog']`	`'dog'`
`s.push(True)`	`[4,'dog',True]`	
`s.size()`	`[4,'dog',True]`	`3`
`s.is_empty()`	`[4,'dog',True]`	`False`
`s.push(8.4)`	`[4,'dog',True,8.4]`	
`s.pop()`	`[4,'dog',True]`	`8.4`
`s.pop()`	`[4,'dog']`	`True`
`s.size()`	`[4,'dog']`	`2`

Stacks

- Stack application

balanced	vs.	unbalanced
(( ) ( ) ( ) ( ))		((((((( ))
(((( ))))		( )))
(( ) ((( )) ( )))		(( ) ( ) (( )

Stack application

(( ) ( ) ( ) ( ))

((((((( ))

```
1  import Stack #import the Stack class as previously defined
2
3  def par_checker(symbol_string):
4      s = Stack()
5      balanced = True
6      index = 0
7      while index < len(symbol_string) and balanced:
8          symbol = symbol_string[index]
9          if symbol == "(":
10             s.push(symbol)
11         else:
12             if s.is_empty():
13                 balanced = False
14             else:
15                 s.pop()
16
17         index = index + 1
18
19     if balanced and s.is_empty():
20         return True
21     else:
22         return False
23
24 print(par_checker('((()))'))
25 print(par_checker('(()'))
```

Queue

Queues

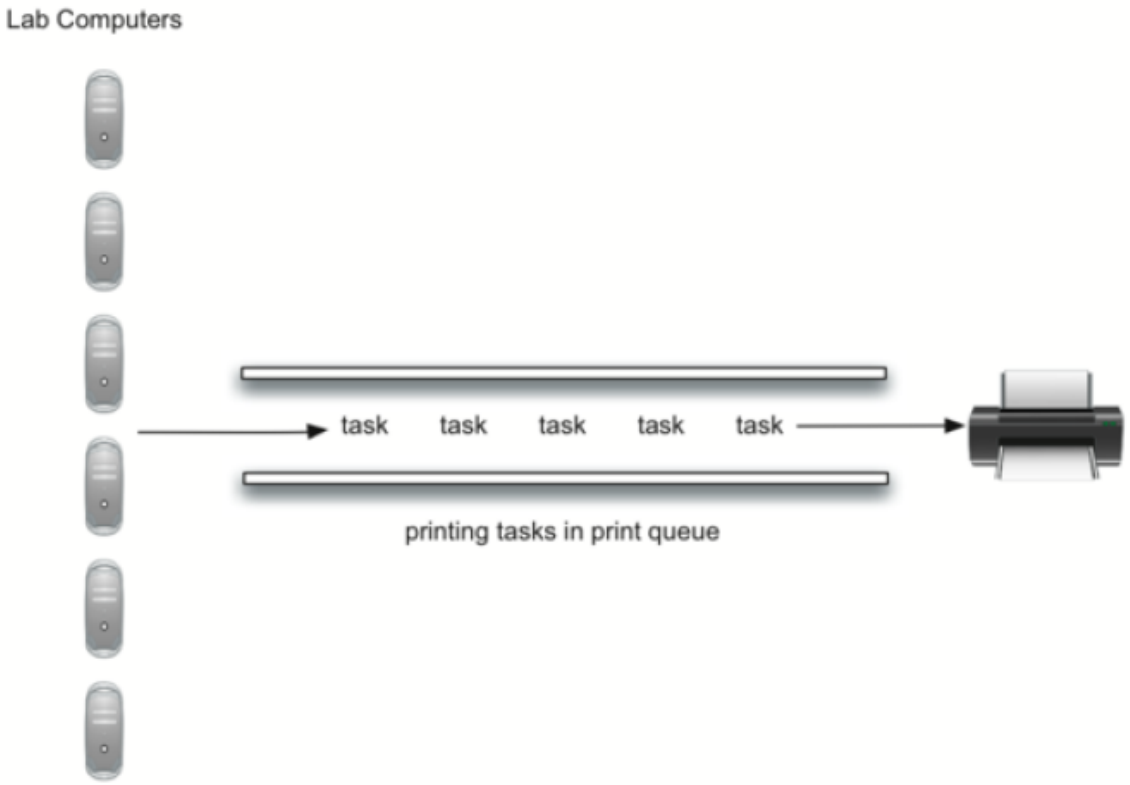
- Example:

Queues

- A queue is an ordered collection of items where the addition of new items happens at one end (rear) and the removal of existing items occurs at the other (front).

Queues

- Queues are first-in first-out (FIFO)
  - First come first served



Queues

- Queue abstract data type:
  - 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.

Queue class
implementation

```
# 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)
```

Queue

- Queue operation examples:

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`

Queues

- Queue application

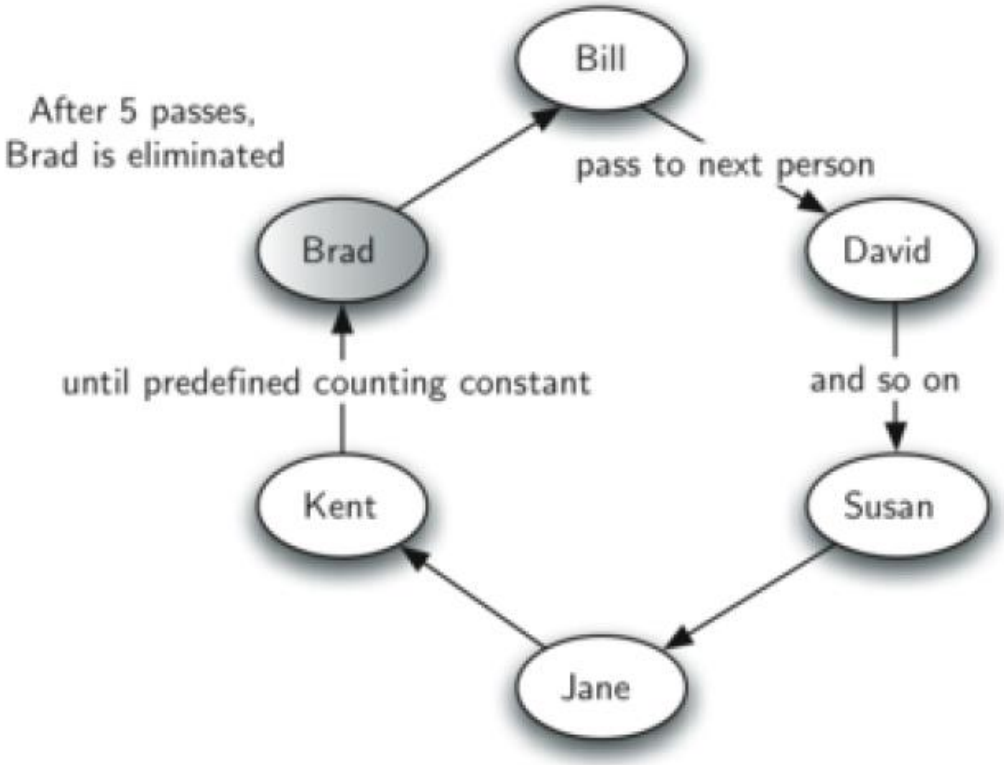


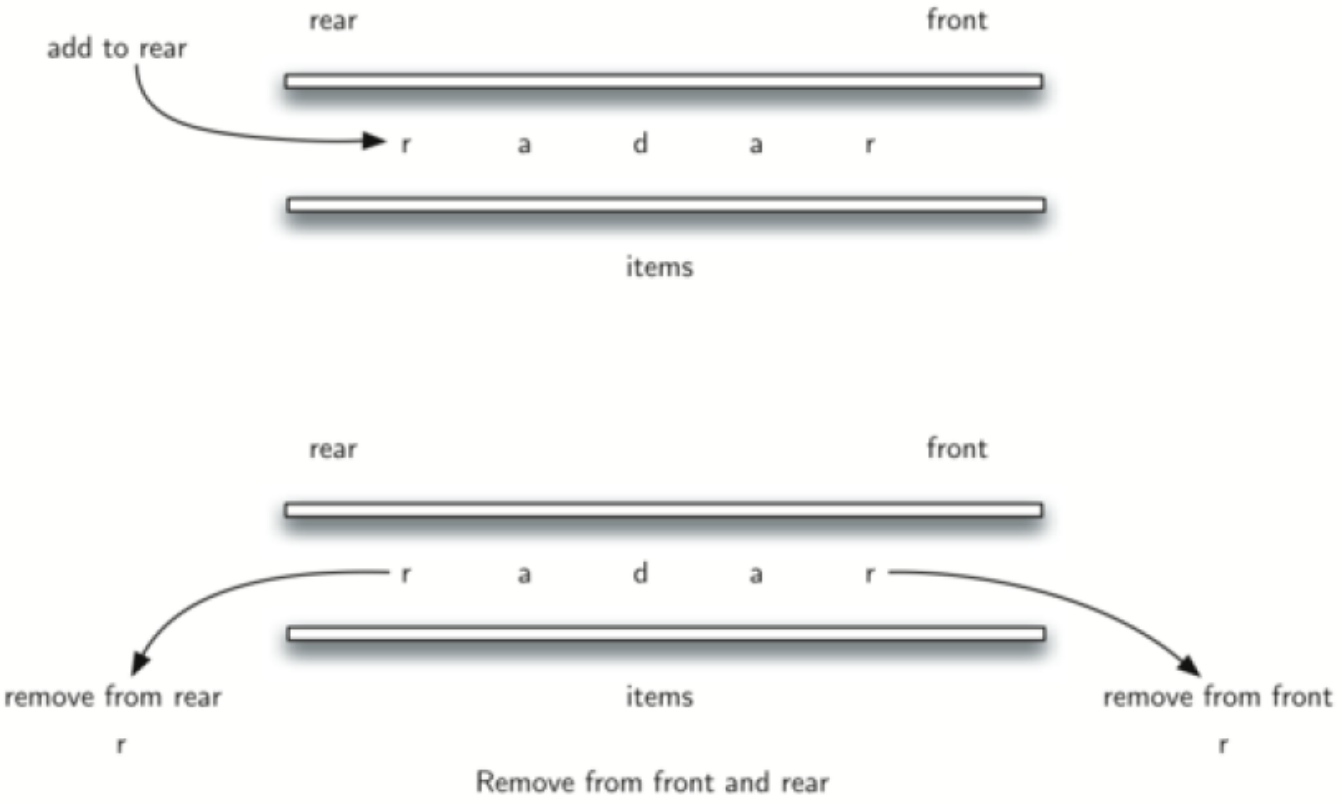
Figure 3.13: A Six Person Game of Hot Potato)

Queue application

```
import Queue  # As previously defined

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

    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))
```

Deques

Deques


rear
front
add to rear
r a d a r
items
rear
front
r a d a r
remove from rear
r
items
remove from front
r
Remove from front and rear

Deques

- A deque, or double-ended queue, is an ordered collection of items similar to the queue:
  - Two ends, front and rear
  - Items can be added and removed from either front and rear
- A deque has all capabilities of a stack and a queue

Deques

- Deques are no longer restricted to LIFO and FIFO
- Consistent use of addition and removal in the context of application

Deques

- Deque abstract data type:
  - 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.
  - add_rear(item) adds a new item to the rear of the deque. It needs the item and returns nothing.

Deques

- Deque abstract data type:
  - 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.

Deque class implementation

```
# 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)
```

Deques

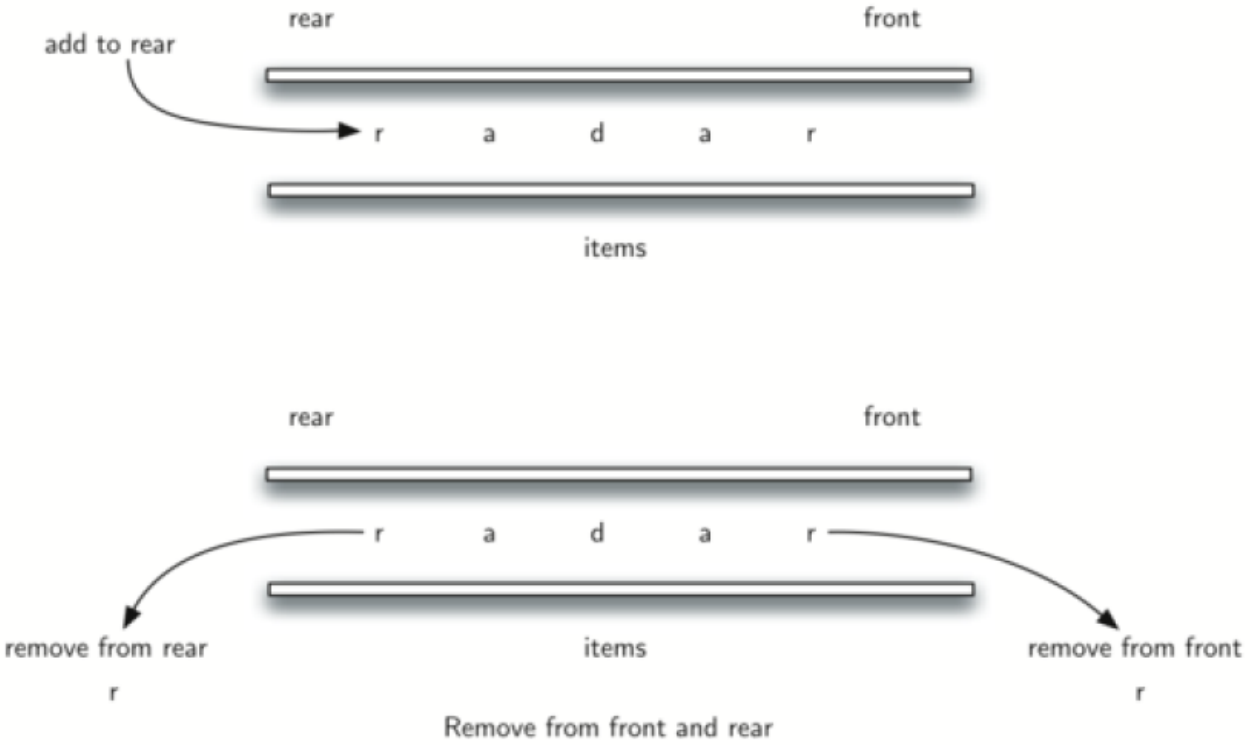
- Deque operation examples:

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

Deques

- Deque application



Deque application

```
import Deque  # As previously defined

def pal_checker(a_string):
    char_deque = Deque()

    for ch in a_string:
        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"))
```

References

- **Chapter 1 Introduction**, of Miller and Ranum (2013). Problem Solving with Algorithms and Data Structures using Python. http://interactivepython.org/runestone/static/pythonds/index.html
- **Chapter 3 Data Structures**, of Miller and Ranum (2013). Problem Solving with Algorithms and Data Structures using Python. http://interactivepython.org/runestone/static/pythonds/index.html