Data Structures

1. Linked List

Generally, this data structure contains a data and a link to the next data. This data structure is defined with a “node” class and a “linked_list” class. Compared to a basic array data structure, Linked List is flexible, i.e, it is not bound by a fixed size and can increase its length.

Code snippet :

```
class node(object):
    def __init__(self, data, n = None):
        self.data = data
        self.next = n

class linked_list(object):
    def __init__(self):
        self.root = None
    def add_beg(self, data):
        new_node = node(data, self.root)
        self.root = new_node
    def remove(self, data):
        temp = prev = self.root
        found = False
        while temp.next:
            if temp.data == data:
                found = True
                break
            prev = temp
            temp = temp.next
        if found == False:
            print("\n[*] Given data not found in Linked List")
        else:
            prev.next = temp.next
            temp = None
            print("\n[*] Node successfully removed")
```

Full code at :
https://github.com/AmunRha/ChallengeSet1/blob/master/DataStructures/lnkList.py

2. Doubly Linked List

Doubly Linked List is just like Linked List except that an additional link called “prev” is present which links the every node to its previous node and not just the node next to it. This takes more storage space than Linked List but is highly traversable.

Code snippet :

```
class node(object):
    def __init__(self, data, n1 = None, n2 = None):
        self.data = data
        self.next = n1
        self.prev = n2

class linked_list(object):
    def __init__(self):
        self.root = None
    def add_beg(self, data):
        new_node = node(data, self.root)
        if self.root is not None:
            self.root.prev = new_node
        self.root = new_node
    def remove(self, data):
        temp = self.root
        found = False
        while temp.next:
            if temp.data == data:
                found = True
                break
            temp = temp.next
        if found == True:
            temp.next.prev = temp.prev
            temp.prev.next = temp.next
            temp = None
            print("\n[*] Node successfully removed")
        elif found == False:
```

```
            print("\n[*] Given data not found in Linked List")
```

Full code at :
https://github.com/AmunRha/ChallengeSet1/blob/master/DataStructures/doublyLnkList.py

3. Circular Linked List

Circular Linked List is just like Linked List except that the last node is connected to the root/first node which makes a cycle or a loop.

Code snippet :

```
class node(object):
    def __init__(self, data, n = None):
        self.data = data
        self.next = n

class linked_list(object):
    def __init__(self):
        self.root = None
    def add(self, data):
        new_node = node(data)
        if self.root is None:
            new_node.next = new_node
            self.root = new_node
        else:
            new_node.next = self.root
            temp = self.root
            while temp.next is not self.root:
                temp = temp.next
            temp.next = new_node
            self.root = new_node
    def remove(self, data):
        temp = prev = self.root
        found = False
        while temp.next is not self.root:
            if temp.data == data:
                found = True
```

```
                break
            prev = temp
            temp = temp.next
        if found == False:
            print("\n[*] Given data is not in the Linked List")
        elif found == True:
            prev.next = temp.next
            temp = None
            print("\n[*] Node successfully removed")
```

Full code at :
https://github.com/AmunRha/ChallengeSet1/blob/master/DataStructures/cirLnkList.py

4. Stack

It is one of the fundamental Data Structures that are implemented across various places. It works under the principle of **"Last In First Out"**. The data is "pushed" into the stack, so every subsequent data is on top of the previous pushed data. When a data is retrieved, it is "popped" from the stack, thus the top most data (recently added data) is retrieved from the stack. In python, it is implemented using in-built list functions,

```
list.append(data)
list.pop()
```

Code snippet :

```
class stack(object):
    def __init__(self):
        self.stack = []
    def push_stack(self, data):
        self.stack.append(data)
    def pop_stack(self):
        try:
            self.stack.pop()
        except:
            print("\n[*] No more elements to remove from stack")
```

Full code at :
https://github.com/AmunRha/ChallengeSet1/blob/master/DataStructures/stack.py

5. Queue

This is also one of the fundamental Data Structures. It works under the principle of **"First In First Out"**. The data is added into the queue through its "rear" part, and the data is retrieved from the queue through the "front" part, thus the first data is always retrieved. In python, it is implemented using in-built functions,

```
list.append(data)
list.pop(0)
```

Code snippet :

```
class queue(object):
    def __init__(self):
        self.queue = []
    def push_queue(self, data):
        self.queue.append(data)
    def pop_queue(self):
        try:
            self.queue.pop(0)
        except:
            print("\n[*] Queue is empty")
```

Full code at :
https://github.com/AmunRha/ChallengeSet1/blob/master/DataStructures/queue.py

6. Double Ended Queue

In a Double Ended Queue, data can be added either through the rear and retrieved from the front or added through the front and retrieved from the rear. In python, it is implemented using in-built functions,

```
list.append(data)
list.pop(0)
list.insert(0,data)
list.pop()
```

Code snippet :

```
class dequeue(object):
```

```
    def __init__(self):
        self.deq = []
    def push_front(self, data):
        self.deq.append(data)
    def pop_rear(self):
        try:
            self.deq.pop(0)
        except:
            print("[*] Double ended queue is empty")
    def push_rear(self, data):
        self.deq.insert(0, data)
    def pop_front(self):
        self.deq.pop()
```

Full code at :
https://github.com/AmunRha/ChallengeSet1/blob/master/DataStructures/dequeue.py