Section A4 Stack and Queue

Stack

Stack Operations

There are different operations associated with the stack data structure. These include,

- ***empty()*** - checking whether a stack is empty
- ***size()*** - returning the number of data elements in a stack
- ***top()*** - returning the top-most data element in a stack
- ***push(element)*** - adding the value passed as a parameter to a stack
- ***pop()*** - deleting the top-most data element in a stack

Stack Implementation

There are different ways that you could use to implement a stack in Python. These include,

- using lists
- using collections.deque
- using queue.LifoQueue
- using singly linked lists

Using Lists

lists can be used as a stack. In lists, append() mimics the push() operation, which is used to add elements to the top of the stack, while the pop() method removes the element in LIFO order.

```
stack = []

#empty() operation
#use to check if the stack is empty
print(not bool(stack))
```

```
#top() operation
#get the top of the stack
print('\nStack top element')
print(stack[-1]) #using backward indexing
print(stack[len(stack)-1])

#pop() operation
#pop() to pop element from stack in LIFO order
print('\nElements popped from stack:')
print(stack.pop())
print(stack.pop())
print(stack.pop())
```

The biggest shortcoming is that it can run into speed problems as the stack grows. The elements in the list are stored next to each other in the computer's

memory. Therefore, if the stack grows bigger (than the block of memory that currently holds it), then Python needs to perform some memory allocations, which could lead to some delays in append() calls.

Using collections.deque

Python provides a container class used to store collections of data named the 'collection' module. In the collection module, there is a *deque* class that we can use to implement stacks. The general syntax to do is as follows.

```
from collections import deque
stack = deque()
```

The use of collections.deque is preferred over the list in the cases where you need quicker append and pop operations. The standard methods in a stack using collections.deque are the same as a list.

```
#pushing elements to stack
stack.append(1)
stack.append(2)
stack.append(3)
stack.append(4)
stack.append(5)
print("Stack:",list(stack))

#to check if stack is empty
print("Is the stack empty? T/F", not bool(stack))

#to find the size of the stack
print("Size is",len(stack))

#to print the topmost element
print("Top is",stack[-1])

#to pop an element
print(stack[-1],"is popped")
stack.pop()
print("Stack:",list(stack))
```

Using Queue Module

The queue module in Python contains many classes used for implementing multi-producer and multi-consumer queues that we can use for parallel computing. The queue module has a Last-In-First-Out Queue, which is a Stack.

```
from queue import LifoQueue
stack = LifoQueue(maxsize=0)
```

In the *maxsize* parameter, we need to mention an integer that sets the upper bound limit on the number of elements that can be placed. Insertion will be blocked once the specified size has been reached. If *maxsize* is less than or

equal to zero, the stack size is infinite.

```
from queue import LifoQueue

# Initializing a stack
stack = LifoQueue(maxsize=3)

#qsize() show the number of elements in the stack
print(stack.qsize())

# put() function to push element in the stack
stack.put(1)
stack.put(2)
stack.put(3)

print("Full: ", stack.full())
print("Size: ", stack.qsize())

# get() function to pop element from stack in LIFO order
print('\nElements popped from the stack')
print(stack.get())
print(stack.get())
print(stack.get())

print("\nEmpty: ", stack.empty())
```

Let's now have a look at the functions available in the queue module:

- ***maxsize* -** number of elements allowed.
- ***empty()*** - return True if the queue is empty, otherwise, False.
- ***full()*** - return True if there are *maxsize* elements. If initialised with maxsize=0 (the default), then full() will never return True.
- ***get()*** - remove and return an element (like pop()). If empty, wait until an element is available.
- ***get_nowait()*** - return an element if one is immediately available. Oherwise, raise QueueEmpty.
- ***put(element)*** - Put an element like push(). If the queue is full, wait until a free slot is available before adding the element.
- ***put_nowait(element)*** - put an element without blocking.
- ***qsize()*** - return the number of elements. If no free slot is immediately available, raise QueueFull.

Using singly linked lists

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class Stack:
    #head is default NULL
    def __init__(self):
        self.head = None

    #checks if stack is empty
    def isempty(self):
        if self.head == None:
            return True
        else:
            return False

    #method to add data to the stack (adds to the start of the stack)
    def push(self,data):

        if self.head == None:
            self.head=Node(data)

        else:
            newnode = Node(data)
            newnode.next = self.head
            self.head = newnode

    #remove element that is the current head (start of the stack)
    def pop(self):

        if self.isempty():
            return None
        else:
            # Removes the head node and makes the preceding one the new head
            poppednode = self.head
            self.head = self.head.next
            poppednode.next = None
            return poppednode.data

    #returns the head node data
    def peek(self):

        if self.isempty():
            return None
        else:
            return self.head.data

    #prints out the stack
    def display(self):

        iternode = self.head
        if self.isempty():
            print("Stack Underflow")
        else:
            while(iternode != None):
                print(iternode.data,"->",end = " ")
                iternode = iternode.next
            return
```

Queue

In a queue data structure, we maintain two pointers, front and rear.

- The **rear** represents the point where the elements are inserted inside a queue.
- The **front** represents the point where the elements from a queue will be removed.

In queues, ***Enqueue*** is the operation of adding a new element after the rear and moves rear to the next element. ***Dequeue*** is the operation of removing the front

element and moving the front to the next element.

Queue Operations
There are different operations associated with the queue data structure. These include,

- ***enqueue(element):*** adding an element to a queue. If the queue is full, then it reaches an overflow condition.
- ***dequeue():*** removing an element from a queue. The elements are popped in the same order in which they are inserted. If the queue is empty, then it reaches an underflow condition.
- ***front():*** getting the front element from a given queue.
- ***rear():*** getting the last element from a given queue.
- ***empty():*** checking whether a queue is empty.
- ***size():*** returning the number of data elements in a queue.

Queue Implementation
Like we saw in stacks, there are different ways that you could use to implement a queue in Python. These include,

- using lists
- using collections.deque
- using queue.Queue
- using singly linked lists

Using Lists
Same as in stacks, we could use append() and pop() functions to mimic the behaviours of enqueue() and dequeue().

```
class Queue:

    def __init__(self):
        self.elements = []

    def enqueue(self, data):
        self.elements.append(data)
        return data

    def dequeue(self):
        return self.elements.pop(0)

    def rear(self):
        return self.elements[-1]

    def front(self):
        return self.elements[0]

    def is_empty(self):
        return len(self.elements) == 0

queue = Queue()
```

using lists is quite slow since inserting or deleting a data element at the beginning of a queue requires shifting all of the other elements in the queue by one (O(n) time).

Using collections.deque

In the collection module, there is a *deque* class that we can use to implement queues. To mimic the behaviours of enqueue and dequeue, we use the functions, append() and popleft() in deque.

```
from collections import deque

#initialising a queue
q = deque()

#adding elements to a queue
q.append('1')
q.append('2')
q.append('3')

print("Initial queue")
print(q)

#removing elements from a queue
print("\nElements dequeued from the queue")
print(q.popleft())
print(q.popleft())
print(q.popleft())

print("\nQueue after removing elements")
print(q)

#uncommenting q.popleft() will raise an IndexError as queue is now empty
q.popleft()
```

Using Queue Module

The implementation of queues using the queue module is similar to stacks. The only difference is instead of using the from queue import LifoQueue, we use from queue import Queue. This Queue follows the FIFO rule. queue.Queue(*maxsize*) initialises a variable to a maximum size of maxsize. A *maxsize* of '0' means an infinite queue.

```
from queue import Queue

#initialising a queue
q = Queue(maxsize = 3)

#qsize() give the maxsize of the Queue
print(q.qsize())

#adding of element to queue
q.put(1)
q.put(2)
q.put(3)

#return Boolean for Full Queue
print("\nFull: ", q.full())

#removing element from queue
print("\nElements dequeued from the queue")
print(q.get())
print(q.get())
print(q.get())

#return Boolean for Empty queue
print("\nEmpty: ", q.empty())

q.put(1)
print("\nEmpty: ", q.empty())
print("Full: ", q.full())

#this would result into Infinite Loop as the Queue is empty.
#print(q.get())
```

Using Singly Linked Lists

```
class Node:

    def __init__(self, data):
        self.data = data
        self.next = None

class Queue:

    def __init__(self):
        self.front = None
        self.rear = None

    def is_empty(self):
        return self.front == None

    def enqueue_operation(self, item):
        current_node = Node(item)

        if self.rear == None:
            self.front = self.rear = current_node
            return
        self.rear.next = current_node
        self.rear = current_node

    def dequeue_operation(self):

        if self.is_empty():
            return
        current_node = self.front
        self.front = current_node.next

        if(self.front == None):
            self.rear = None
```