Experiment No. 7
Program for data structure using built in function for link list, stack and queues
Date of Performance:
Date of Submission:

Experiment No. 7

Title: Program for data structure using built in function for link list, stack and queues

Aim: To study and implement data structure using built in function for link list, stack and queues

Objective: To introduce data structures in python

Theory:

Stacks -the simplest of all data structures, but also the most important. A stack is a collection of objects that are inserted and removed using the LIFO principle. LIFO stands for "Last In First Out". Because of the way stacks are structured, the last item added is the first to be removed, and vice-versa: the first item added is the last to be removed.

Queues – essentially a modified stack. It is a collection of objects that are inserted and removed according to the FIFO (First In First Out) principle. Queues are analogous to a line at the grocery store: people are added to the line from the back, and the first in line is the first that gets checked out – BOOM, FIFO!

Linked Lists

The Stack and Queue representations I just shared with you employ the python-based list to store their elements. A python list is nothing more than a dynamic array, which has some disadvantages.

The length of the dynamic array may be longer than the number of elements it stores, taking up precious free space.

Insertion and deletion from arrays are expensive since you must move the items next to them over

Using Linked Lists to implement a stack and a queue (instead of a dynamic array) solve both of these issues; addition and removal from both of these data structures (when implemented with a linked list) can be accomplished in constant O(1) time. This is a HUGE advantage when dealing with lists of millions of items.

Linked Lists – comprised of 'Nodes'. Each node stores a piece of data and a reference to its next and/or previous node. This builds a linear sequence of nodes. All Linked Lists store a head, which is a reference to the first node. Some Linked Lists also store a tail, a reference to the last node in the list.

Code:

1) Linked List:

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


class LinkedList:
    def __init__(self):
        self.head = None


    # Traversing the linked list and printing elements with their indices
   def traverse_with_index(self):
        current =
       self.head  index =
       0  while current:
            print("Index:", index, "Data:", current.data)
```

current =

current.next index

+= 1

Appending an element at the end of the linked

list def append(self, data): new_node =

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

new_node

return last_node

= self.head while

last_node.next:

last_node = last_node.next

last_node.next = new_node

Inserting an element at a specific index

def insert_at_index(self, index, data):

new_node =

Node(data) if index ==

0:

new_node.next = self.head

self.head = new_node

```
        return  current
    = self.head
    position = 0

     while current and position < index - 1:
        current =
    current.next  position
    += 1  if current is None:
        print("Index out of
    range.")  return
    new_node.next = current.next
    current.next = new_node


 # Removing an element at a specific index
def remove_at_index(self, index):
     if index == 0:
        if self.head is None:  print("List is
    empty.")  return  self.head =
    self.head.next  return  current =
    self.head  position = 0  while current
    and position < index - 1:
        current = current.next  position += 1
    if current is None or current.next is None:
```

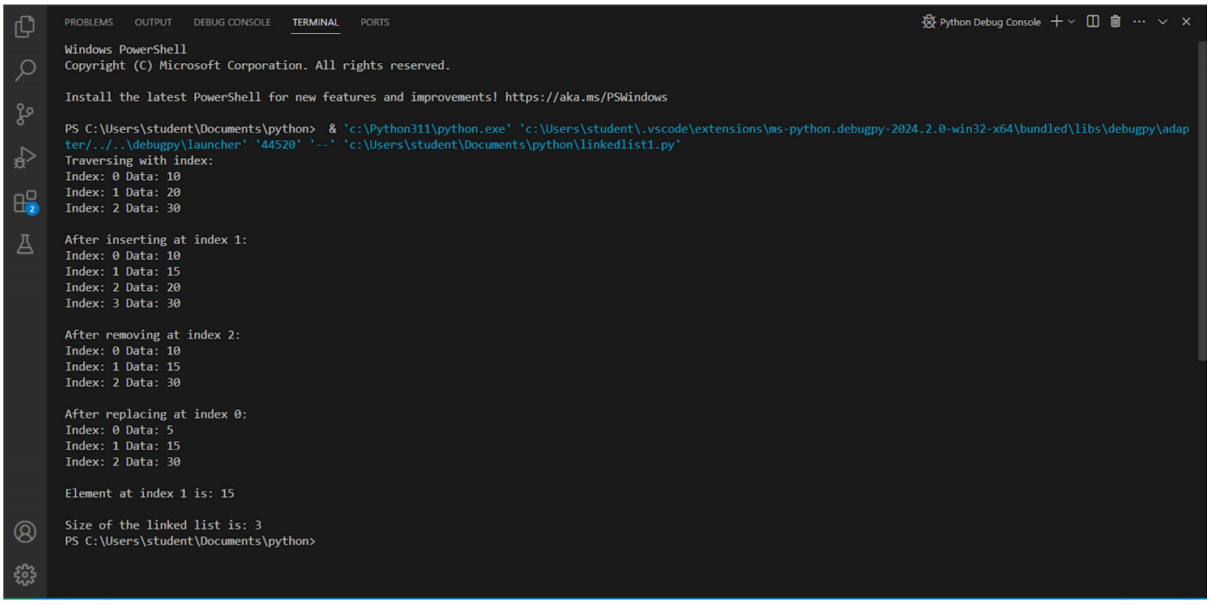
```
   print("Index out of range.")  return
   current.next = current.next.next

 # Replacing an element at a specific index
def replace_at_index(self, index, data):
    current = self.head  position = 0
   while current and position <
   index:
       current =
   current.next  position
   += 1  if current is None:
       print("Index out of
   range.")  return  current.data
   = data

 # Searching for the location of an element by its index
def search_by_index(self, index):
    current = self.head  position = 0
   while current and position <
   index:
       current =
      current.next  position
      += 1
```

```
if current is None:
    print("Index out of
range.")  return  return
current.data, position

# Size of the linked list
def size(self):  count = 0
current = self.head  while
current:  count += 1
current = current.next
return count

# Example usage  if
__name__ == "__main__":
linked_list = LinkedList()

# Appending elements
linked_list.append(10)
linked_list.append(20)
linked_list.append(30)
print("Traversing with
index:")
```

```
linked_list.traverse_with_
index()

 # Inserting an element at index 1
linked_list.insert_at_index(1, 15)
print("\nAfter inserting at index 1:")
linked_list.traverse_with_index()

 # Removing an element at index 2
linked_list.remove_at_index(2)
print("\nAfter removing at index 2:")
linked_list.traverse_with_index()

 # Replacing an element at index 0
linked_list.replace_at_index(0, 5)
print("\nAfter replacing at index 0:")
linked_list.traverse_with_index()

 # Searching for the location of an element at index 1
data, position = linked_list.search_by_index(1)
print("\nElement at index 1 is:", data)

 # Size of the linked list
```

```
    print("\nSize of the linked list is:", linked_list.size())
```

Output:



2)Stack and Queue Implementations Using a Linked List:

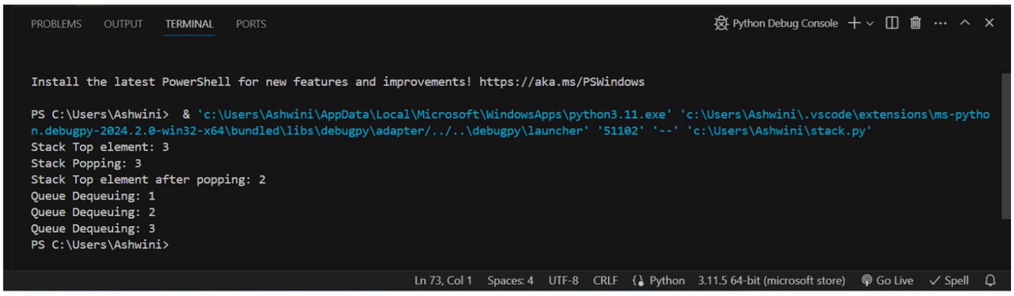
```
class Node:
    def __init__(self, value):
        self.value = value
        self.next = None

class Stack:
    def __init__(self):
        self.top = None
```

```
    def push(self, value):
        new_node = Node(value)
       new_node.next = self.top
       self.top = new_node

    def pop(self):
        if self.is_empty():
            return None  value
       = self.top.value
       self.top = self.top.next
       return value

    def peek(self):
        if self.is_empty():
            return None
       return self.top.value

    def is_empty(self):
        return self.top is None

class Queue:
```

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

def enqueue(self, value):
   new_node = Node(value)
    if not self.rear:
        self.front = self.rear =
  new_node  return  self.rear.next =
  new_node  self.rear = new_node


def dequeue(self):
    if self.is_empty():
        return None  value =
  self.front.value  self.front
  = self.front.next  if not
  self.front:
        self.rear = None
  return value


def is_empty(self):
    return self.front is
   None
```

```
# Example usage:
stack = Stack()
stack.push(1)
stack.push(2)
stack.push(3)

print("Stack Top element:", stack.peek())
print("Stack Popping:", stack.pop())  print("Stack Top
element after popping:", stack.peek())

queue = Queue()
queue.enqueue(1)
queue.enqueue(2)
queue.enqueue(3)

print("Queue Dequeuing:", queue.dequeue())
print("Queue Dequeuing:", queue.dequeue())
print("Queue Dequeuing:", queue.dequeue())
```

Output:



```
PROBLEMS   OUTPUT   TERMINAL   PORTS                Python Debug Console

Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS C:\Users\Ashwini>  & 'c:\Users\Ashwini\AppData\Local\Microsoft\WindowsApps\python3.11.exe' 'c:\Users\Ashwini\.vscode\extensions\ms-python.debugpy-2024.2.0-win32-x64\bundled\libs\debugpy\adapter/../..\debugpy\launcher' '51102' '--' 'c:\Users\Ashwini\stack.py'
Stack Top element: 3
Stack Popping: 3
Stack Top element after popping: 2
Queue Dequeuing: 1
Queue Dequeuing: 2
Queue Dequeuing: 3
PS C:\Users\Ashwini>

Ln 73, Col 1   Spaces: 4   UTF-8   CRLF   Python   3.11.5 64-bit (microsoft store)   Go Live   ✓ Spell
```

Conclusion:

In conclusion, implementing data structures like stacks, queues, and linked lists using built-in functions in Python offers efficient solutions to manage collections of objects. By leveraging the LIFO and FIFO principles, we can handle data insertion and removal seamlessly, reducing overhead and improving performance, especially with large datasets. Utilizing linked lists over dynamic arrays further enhances efficiency by enabling constant-time operations for addition and removal, making them ideal choices for handling millions of items efficiently.