|  |
| --- |
| 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()

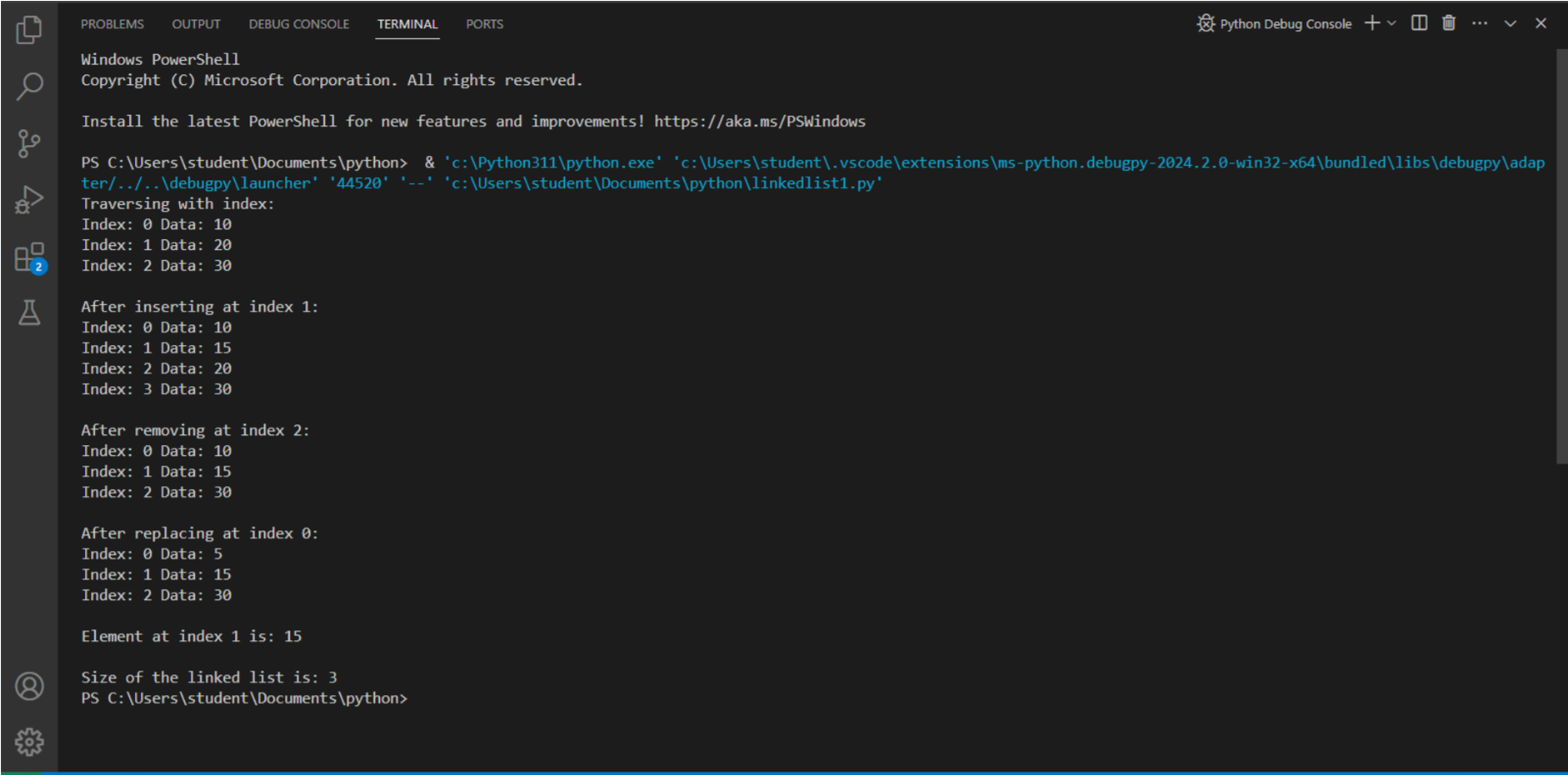
# Removing an element at index 2 linked\_list.remove\_at\_index(2) print("\nAfter removing at index 2:") linked\_list.traverse\_with\_index()

# 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):

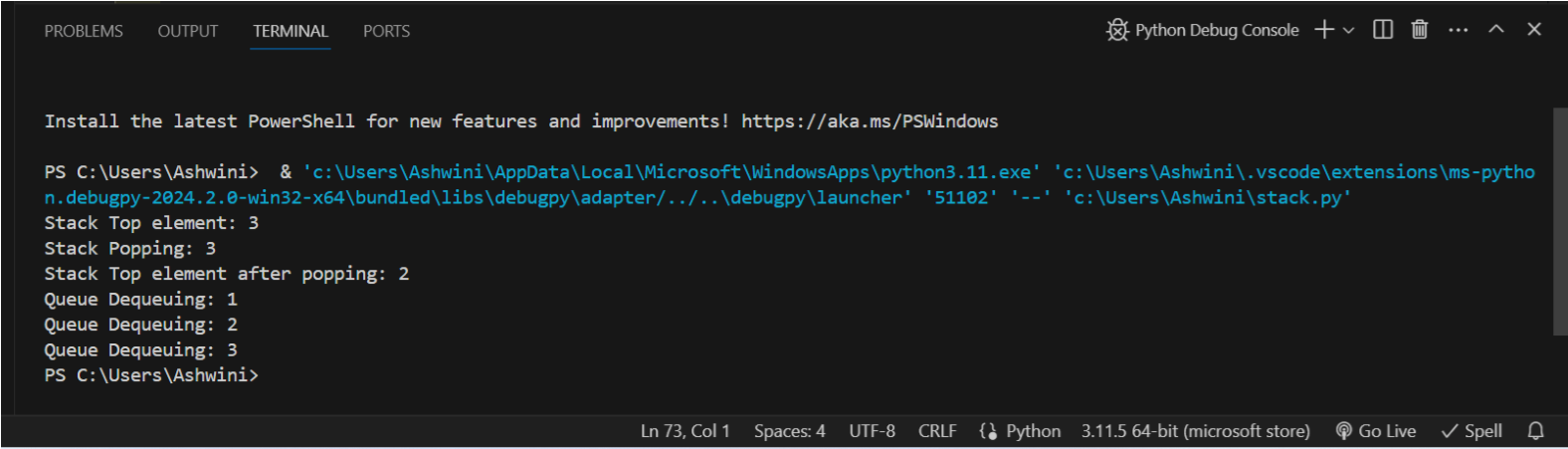
return self.front is None

# 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:**



**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.