|  |
| --- |
| Experiment No. 7 |
| Program for data structure using built in function for link list, stack and queues |
| Date of Performance:16/02/24 |
| Date of Submission:23/02/24 |

**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:**

**#Stack**

st = [3, 5, 13, 14, 6]

n = len(st)

print("Original Stack:", st)

# Functions on Stack

st.append(56) # Inserting element 32

print("Element added to the stack:", st)

popped\_element = st.pop()

print("Popped element from the stack:", popped\_element)

top\_element = st[n-1]

print("Topmost element of the stack:", top\_element)

element\_index = st.index(13)

print("Index of element 21 in the stack:", element\_index)

# Checking if stack is empty or not

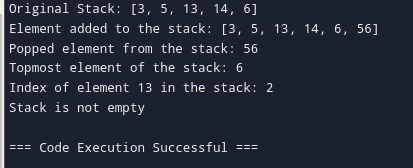
if st == []:

print("Stack is empty")

else:

print("Stack is not empty")

**Output:**



**# Linked List**

ls = [3, 5, 13, 14, 6]

print("Original Linked List:", ls)

# Operations on Linked List

# Traversing

print("Elements in the Linked List:")

for element in ls:

print(element)

ls.append(56) # Appending 227 at the end

print("After appending 56 at the end:", ls)

ls.insert(3, 8) # Inserting 31 at position 3

print("After inserting 8 at position 3:", ls)

ls.remove(56) # Removing element 4

print("After removing element 56:", ls)

# Replacing an element

ls.remove(13)

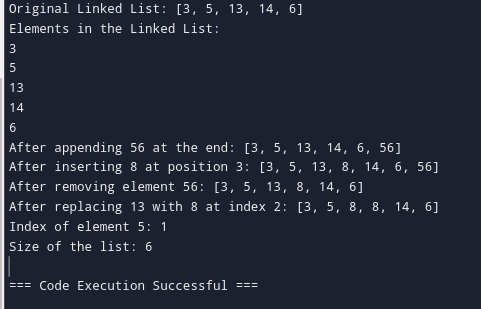
ls.insert(2, 8)

print("After replacing 13 with 8 at index 2:", ls)

print("Index of element 5:", ls.index(5)) # Searching for element 5

print("Size of the list:", len(ls)) # Size of the list

**Output:**



**#Queue**

class Queue:

def \_\_init\_\_(self):

# Initialize an empty list to store the elements of the queue

self.que = []

def isempty(self):

# Check if the queue is empty

return self.que == []

def add(self, element):

# Add an element to the end of the queue

self.que.append(element)

def delete(self):

# Delete and return the element at the front of the queue

if self.isempty():

# If the queue is empty, return -1 indicating underflow

return -1

else:

# Otherwise, remove and return the first element

return self.que.pop(0)

def search(self, element):

# Search for an element in the queue and return its position

if self.isempty():

# If the queue is empty, return -1 indicating underflow

return -1

else:

try:

# Attempt to find the index of the element

n = self.que.index(element)

# If found, return its position (1-indexed)

return n + 1

except ValueError:

# If element not found, return -2 indicating element not found

return -2

def display(self):

# Display the elements of the queue

return self.que

# Test the Queue class

q = Queue()

q.add(3)

q.add(5)

q.add(13)

q.add(14)

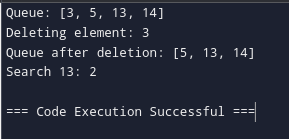
print("Queue:", q.display())

print("Deleting element:", q.delete())

print("Queue after deletion:", q.display())

print("Search 13:", q.search(13))

**Output:**



**Conclusion:** In this program, we explored and implemented fundamental data structures using Python's built-in functionalities. We focused on three key data structures:

1] Stacks: We learned that stacks operate on the Last In, First Out (LIFO) principle, where the last item added is the first to be removed. This data structure is essential for various applications such as function calls, expression evaluation, and backtracking algorithms.

2] Queues: Queues follow the First In, First Out (FIFO) principle, resembling a line at a grocery store where the first person to join the line is the first to be served. Queues are widely used in scenarios like task scheduling, breadth-first search algorithms, and message processing systems.

3] Linked Lists: Unlike arrays, linked lists offer dynamic memory allocation and efficient insertion and deletion operations at the cost of sequential access. Each node in a linked list contains data and a reference to the next node, forming a linear sequence. Linked lists are versatile and find applications in implementing stacks, queues, as well as more complex data structures like graphs and trees.