|  |
| --- |
| Experiment No. 7 |
| Program for data structure using built in function for link list, stack and queues |
| Date of Performance: 28/02/2024 |
| Date of Submission: 13/03/2024 |

**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.

**1] Stack Program:**

# Stack operations

stack = [1, 2, 21, 4,66]

stack\_length = len(stack)

print("Original Stack:", stack)

# Pushing an element onto the stack

stack.append(32)

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

# Popping an element from the stack

popped\_element = stack.pop()

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

# Accessing the topmost element of the stack

top\_element = stack[-1]

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

# Finding the index of an element in the stack

element\_index = stack.index(21)

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

# Checking if the stack is empty

if not stack:

print("Stack is empty")

else:

print("Stack is not empty")

**Output:**

Original Stack: [1, 2, 21, 4, 66]

Element added to the stack: [1, 2, 21, 4, 66, 32]

Popped element from the stack: 32

Topmost element of the stack: 66

Index of element 21 in the stack: 2

Stack is not empty

**2] Linked List Program:**

# Linked List operations

linked\_list = [4, 3, 1, 5, 21]

print("Original Linked List:", linked\_list)

# Traversing the Linked List

print("Elements in the Linked List:")

for element in linked\_list:

print(element)

# Appending an element at the end

linked\_list.append(227)

print("After appending 227 at the end:", linked\_list)

# Inserting an element at a specific position

linked\_list.insert(3, 31)

print("After inserting 31 at position 3:", linked\_list)

# Removing an element

linked\_list.remove(4)

print("After removing element 4:", linked\_list)

# Replacing an element

linked\_list.remove(1)

linked\_list.insert(2, 121)

print("After replacing 1 with 121 at index 2:", linked\_list)

# Searching for an element and finding its index

print("Index of element 5:", linked\_list.index(5))

# Finding the size of the list

print("Size of the list:", len(linked\_list))

**Output:**

Original Linked List: [4, 3, 1, 5, 21]

Elements in the Linked List:

4

3

1

5

21

After appending 227 at the end: [4, 3, 1, 5, 21, 227]

After inserting 31 at position 3: [4, 3, 1, 31, 5, 21, 227]

After removing element 4: [3, 1, 31, 5, 21, 227]

After replacing 1 with 121 at index 2: [3, 31, 121, 5, 21, 227]

Index of element 5: 3

Size of the list: 6

1. **Queue Program:**

class Queue:

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

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

self.elements = []

def is\_empty(self):

# Check if the queue is empty

return not self.elements

def enqueue(self, element):

# Add an element to the end of the queue

self.elements.append(element)

def dequeue(self):

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

if self.is\_empty():

# If the queue is empty, return None indicating underflow

return None

else:

# Otherwise, remove and return the first element

return self.elements.pop(0)

def search(self, element):

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

if self.is\_empty():

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

return -1

else:

try:

# Attempt to find the index of the element

index = self.elements.index(element)

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

return index + 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.elements

# Test the Queue class

q = Queue()

q.enqueue(1)

q.enqueue(2)

q.enqueue(3)

q.enqueue(4)

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

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

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

print("Search 3:", q.search(3))

**Output:**

Queue: [1, 2, 3, 4]

Deleting element: 1

Queue after deletion: [2, 3, 4]

Search 3: 2

**Conclusion:**

This experiment delved into fundamental data structures like stacks, queues, and linked lists, employing built-in functions in Python for implementation. Each structure operates on unique principles - stacks follow Last-In-First-Out (LIFO), queues adhere to First-In-First-Out (FIFO), and linked lists utilize node-based sequencing. These structures showcase diverse applications, facilitating efficient data organization and manipulation. By mastering these concepts, programmers can enhance their proficiency and problem-solving skills in Python, contributing to robust software development and algorithmic design.