Experiment No. 6
Program for data structure using built in function for link list, stack and queues
Date of Performance:
Date of Submission:

Experiment No. 6

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.

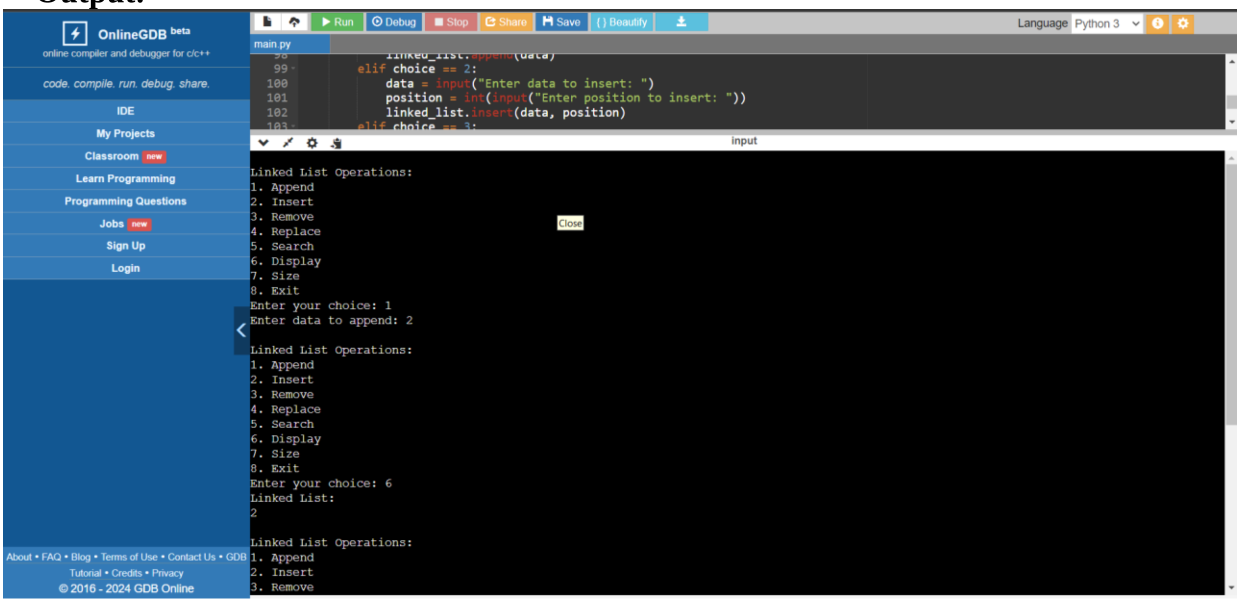
Code:

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

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

    def append(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node

    def insert(self, data, position):
        new_node = Node(data)
        if position == 0:
            new_node.next = self.head
            self.head = new_node
            return
        current_node = self.head
        for _ in range(position - 1):
            if current_node.next:
                current_node = current_node.next
            else:
                raise IndexError("Index out of range")
        new_node.next = current_node.next
        current_node.next = new_node

    def remove(self, data):
        current_node = self.head
        if current_node and current_node.data == data:
            self.head = current_node.next
            current_node = None
            return
        prev_node = None
        while current_node and current_node.data != data:
            prev_node = current_node
            current_node = current_node.next
```

```
        if current_node is None:
            return
        prev_node.next = current_node.next
        current_node = None

    def replace(self, old_data, new_data):
        current_node = self.head
        while current_node:
            if current_node.data == old_data:
                current_node.data = new_data
            current_node = current_node.next

    def search(self, data):
        current_node = self.head
        while current_node:
            if current_node.data == data:
                return True
            current_node = current_node.next
        return False

    def display(self):
        current_node = self.head
        while current_node:
            print(current_node.data, end=" ")
            current_node = current_node.next
        print()

    def size(self):
        count = 0
        current_node = self.head
        while current_node:
            count += 1
            current_node = current_node.next
        return count

if __name__ == "__main__":
    linked_list = LinkedList()

    while True:
        print("\nLinked List Operations:")
        print("1. Append")
        print("2. Insert")
        print("3. Remove")
        print("4. Replace")
        print("5. Search")
        print("6. Display")
        print("7. Size")
        print("8. Exit")

        choice = int(input("Enter your choice: "))

        if choice == 1:
            data = input("Enter data to append: ")
            linked_list.append(data)
        elif choice == 2:
            data = input("Enter data to insert: ")
            position = int(input("Enter position to insert: "))
```

```
        linked_list.insert(data, position)
    elif choice == 3:
        data = input("Enter data to remove: ")
        linked_list.remove(data)
    elif choice == 4:
        old_data = input("Enter data to replace: ")
        new_data = input("Enter new data: ")
        linked_list.replace(old_data, new_data)
    elif choice == 5:
        data = input("Enter data to search: ")
        if linked_list.search(data):
            print("Data found in the linked list.")
        else:
            print("Data not found in the linked list.")
    elif choice == 6:
        print("Linked List:")
        linked_list.display()
    elif choice == 7:
        print("Size of the linked list:", linked_list.size())
    elif choice == 8:
        print("Exiting...")
        break
    else:
        print("Invalid choice. Please try again.")
```

Output:



Conclusion: Data structures python has been studied and implemented.