**Python & Data Structures Laboratory B.Tech. 3rd Semester**



**Name : Somya Jha**

**Roll Number : 23ETMC412012**

**Department : Computer Science and Engineering**

**Faculty of Engineering & Technology**

**Ramaiah University of Applied Sciences**

**Ramaiah University of Applied Sciences**

Private University Established in Karnataka State by Act No. 15 of 2013

|  |  |
| --- | --- |
| Faculty | Engineering & Technology |
| Programme | B. Tech. in Computer Science and Engineering |
| Year/Semester | 2nd year/ 3rd semester |
| Name of the Laboratory | Python & Data Structures Laboratory |
| Laboratory Code | 21CSL206A |

**Index**

|  |  |  |  |
| --- | --- | --- | --- |
| **No** | **Lab Experiment** | **Lasb Documents** | **Viva** |
| 1 | Array | 10% of Lab CE |  |
| 2 | Linked List | 10% of Lab CE |  |
| 3 | Stack | 10% of Lab CE |  |
| 4 | Queue | 10% of Lab CE |  |
| 5 | Binary Tree | 10% of Lab CE |  |
| 6 | Binary Search Tree | 10% of Lab CE |  |
| 7 | Heap | 10% of Lab CE |  |
| 8 | AVL Tree | 10% of Lab CE |  |
| 9 | Quick Sort | 10% of Lab CE |  |
| 10 | Lab Internal | 10% of Lab CE | |

|  |  |  |
| --- | --- | --- |
| **No** | **Experiment Name** | **Page No** |
| 1 | Array | 5 |
| 2 | Linked List | 7 |
| 3 | Stack | 9 |
| 4 | Queue | 11 |
| 5 | Binary Tree | 12 |
| 6 | Binary Search Tree | 14 |
| 7 | Heap | 15 |
| 8 | AVL Tree | 16 |
| 9 | Quick Sort | 17 |

# Experiment 1

**Title of the laboratory experiment**: Array

1. **Aim:**

To understand and implement the basic operations in arrays using python.

1. **Objective:**

To execute the below operations:

1. Traverse − print all the array elements one by one.
2. Insertion − Adds an element at the given index.
3. Deletion − Deletes an element at the given index.
4. Search − Searches an element using the given index or by the value.
5. Update − Updates an element at the given index.
6. **Exercise:**

To develop a python to perform the below tasks:

1. Create your own list of your favourite five sportsperson. Using this find out,
2. Length of the list.
3. Add a sixth sportsperson at the end of this list.
4. You realize that you need to add the sixth sportsperson after the second sportsperson, so remove it from the list first and then add it after the second sportsperson.
5. Now you don't like two sportspersons. Now remove those two and replace them with any other two sportspersons.
6. Sort the sportspersons list in alphabetical order (hint: use the dir() functions to list down all functions available in the list).
7. Create a list of all even numbers between number x and number y. The number x should be your age, and the number y should be your father's or mother's age.
8. **Experimental Procedure**
   1. **Create your own list of your five-favourite sportsperson.** 
      1. **Algorithm design:**

class SportspersonArray:

init

sportspersons = initial list

get length

return len(sportspersons)

add at end

append(sportsperson)

add after second

remove(sportsperson)

insert(2, sportsperson)

replace two sportspersons

for person in remove list

if person in sportspersons

remove(person)

sportspersons.extend

sort\_sportspersons

sportspersons.sort

print\_sportspersons

print Sportspersons

favourite sportspersons = SportspersonArray

(["Hamilton", "Vettel", "Leclerc", "Sainz", "Ricciardo"])

favourite sportspersons.add at end("Yuki")

favourite sportspersons.print sportspersons()

favourite sportspersons.add after second("Yuki")

favourite sportspersons.print sportspersons()

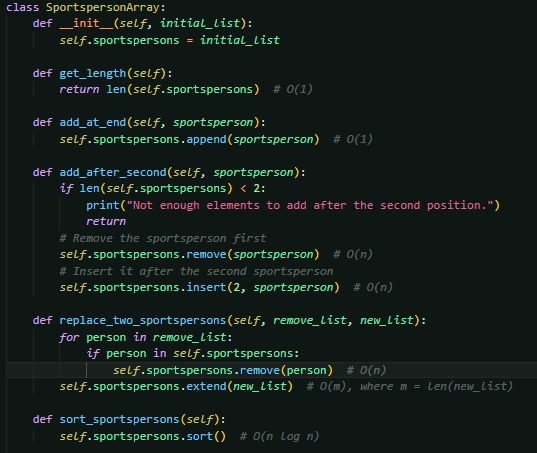
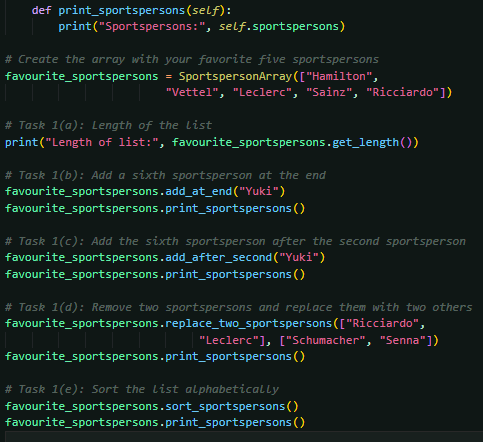
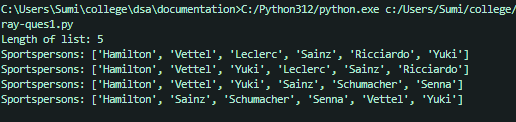
favourite sportspersons.replace two sportspersons

(["Ricciardo", "Leclerc"], ["Schumacher", "Senna"])

favourite sportspersons.print sportspersons()

favourite sportspersons.sort sportspersons()

favourite sportspersons.print sportspersons()

* + 1. **Program:**
    2. **Presentation** **of the results:**
    3. **Analysis and discussions:**

**get\_length()**

**Operation:** Returns the length of the list using len().

**Time Complexity:** O(1)

**add\_at\_end(sportsperson)**

**Operation:** Appends a sportsperson to the end of the list using append().

**Time Complexity:** O(1)

**add\_after\_second(sportsperson)**

**Operation:** Removes the sportsperson using remove(), which involves a linear search. Inserts the sportsperson after the second position using insert(), which shifts elements to the right.

**Time Complexity:** O(n) for remove() and O(n) for insert(), making the overall complexity O(n).

**replace\_two\_sportspersons(remove\_list, new\_list)**

**Operation:** Removes sportspersons using remove() and adds new sportspersons using extend(), which appends all elements of the new, where mmm is the size of the new list).

**Time Complexity:** **O(n\*n)**

**sort\_sportspersons()**

**Operation:** Sorts the list in alphabetical order using sort().

**Time Complexity:** O(n log n)

**print\_sportspersons()**

**Operation:** Prints all elements of the list.

**Time Complexity:** O(n)

|  |  |
| --- | --- |
| **Operation** | **Time Complexity** |
| get\_length() | O(1) |
| add\_at\_end(sportsperson) | O(1) |
| add\_after\_second() | O(n) |
| replace\_two\_sportspersons() | O(n \* n) |
| sort\_sportspersons() | O(n log n) |
| print\_sportspersons() | O(n) |

* 1. **Create a list of all even numbers between number x and number y.**
     1. **Algorithm design:**

class EvenNumbersArray

init

even\_numbers = []

generate even numbers

if x > y:

print x should be less than y.

for num in range(x, y + 1)

if num % 2 == 0:

even numbers.append(num)

print even numbers:

print even numbers

get length:

return len(even numbers)

x = 19

y = 51

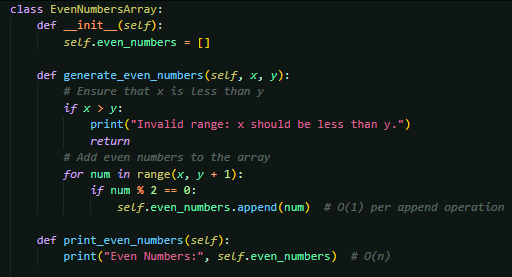
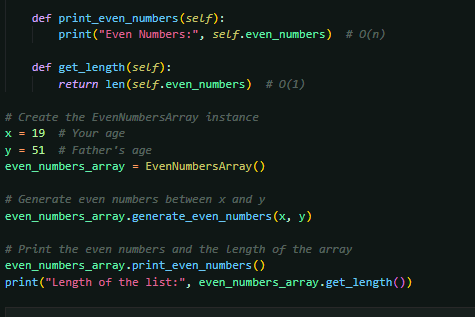
even numbers array = EvenNumbersArray()

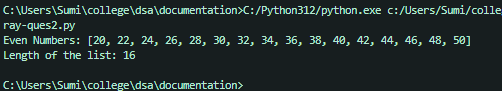
even numbers array.generate even numbers(x, y)

even numbers array.print even numbers()

print even numbers array.get length()

* + 1. **Program:**



* + 1. **Presentation of the results:**
    2. **Analysis and discussions:**

**generate\_even\_numbers(x, y)**

**Operation:** Generates all even numbers between x and y using a for loop. For each number, it checks if the number is even (num % 2 == 0) and appends it to the list if true.

**Time Complexity:** O(n)

**print\_even\_numbers()**

**Operation:** Prints all elements of the array.

**Time Complexity:** O(n)

**get\_length()**

**Operation:** Returns the length of the array using Python's built-in len() function.

**Time Complexity:** O(1)

|  |  |
| --- | --- |
| Operation | Time Complexity |
| |  | | --- | |  |   Generate\_even\_numbers(x, y) | O(n) |
| print\_even\_numbers() | O(n) |
| get\_length() | O(!) |

# Experiment 2

**Title of the Laboratory Exercise**: Linked List

1. **Aim:**

To understand and implement the basic operations in Circular Doubly Linked List using python.

1. **Objective:**

To execute the below operations in Circular Doubly Linked List:

1. Insert: Inserts an element after a specific value.
2. Delete: Deletes an element having a specific value.
3. Display: Prints the elements in the forward direction as well as in the reverse direction.
4. **Exercise:**

In a Circular Doubly Linked List class, implement the below four operations:

**def insert\_after\_value(self, data\_after, data\_to\_insert):**

# Search for first occurance of data\_after value in linked list

# Now insert data\_to\_insert after data\_after node

**def remove\_by\_value(self, data):**

# Remove first node that contains data

**def print\_forward(self):**

# This method prints list in forward direction. Use node.next. Use a print statement to print the nodes in forward direction starting from the first node to the last node.

**def print\_backward(self):**

# Print linked list in reverse direction. Use node.prev for this. Use a print statement to print the nodes in backward direction starting from the last node to the first node.

Now make following calls,

LL = LinkedList()

LL.insert\_values(["Red","Yellow","Purple","Orange"])

LL.print()

LL.insert\_after\_value("Yellow","Blue")

#insert Blue after Yellow

LL.print()

LL.remove\_by\_value("orange")

#remove Orange from linked list

LL.print()

LL.remove\_by\_value("Green")

LL.print()

LL.remove\_by\_value("Red")

LL.remove\_by\_value("Yellow")

LL.remove\_by\_value("Blue")

LL.remove\_by\_value("Purple")

LL.print()

LL.print\_forward()

LL.print\_backward()

**4. Experimental Procedure**

* 1. **Algorithm design**

class Node:

init(self):

data, next, prev

class CircularDoublyLinkedList:

init(self):

head = None

insert\_values:

for data in list:

append(data)

append:

new\_node = Node

if head is None:

head = new\_node

head.next = head

head.prev = head

else:

tail = head.prev

tail.next = new node

new node.prev = tail

new node.next = head

head.prev = new node

insert after value:

current = head

while True:

if current.data == data\_after:

new\_node.next = current.next

new\_node.prev = current

current.next.prev = new node

current.next = new node

return

current = current.next

if current == head:

break

print Value not found

remove by value:

current = head

while True:

if current.data == data:

if current.next == current:

head = None

else:

current.prev.next = current.next

current.next.prev = current.prev

if current == self.head:

self.head = current.next

return

current = current.next

if current == self.head:

break

print Value not found

print forward:

current = head

result = []

while True:

append data

current = current.next

print(result)

def print\_backward:

current = head.prev

result = []

while True:

append(current.data)

current = current.prev

print(result)

LL = CircularDoublyLinkedList()

insert\_values(["Red", "Yellow", "Purple", "Orange"])

print forward()

insert after value("Yellow", "Blue")

print forward()

remove by value("Orange")

print forward()

remove by value("Green")

print forward()

remove by value("Red")

remove by value("Yellow")

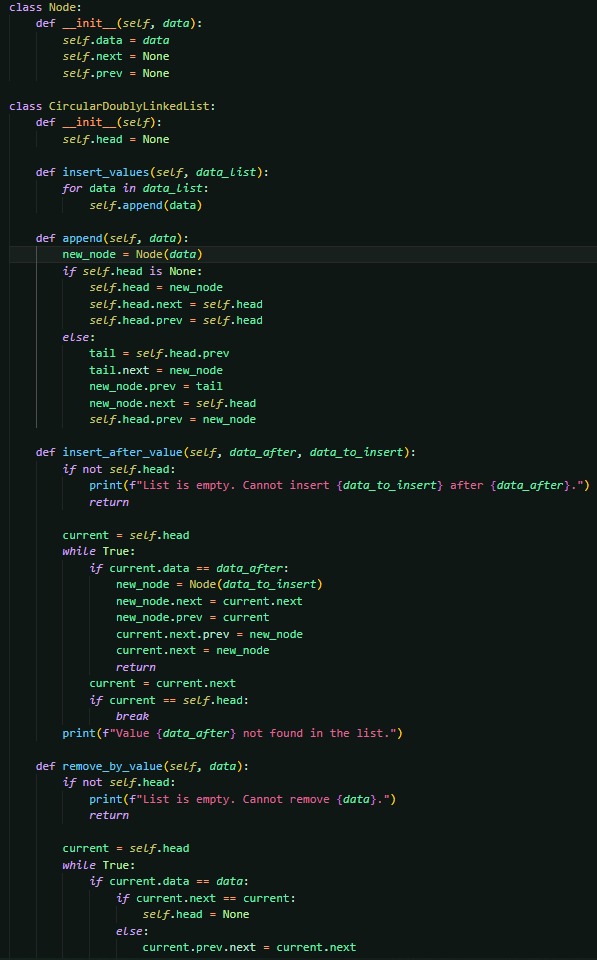
remove by value("Blue")

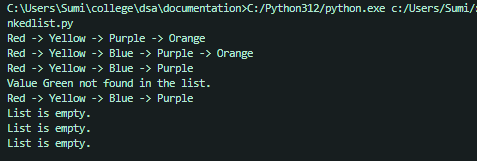
remove by value("Purple")

print forward()

print forward()

print backward()

* 1. **Program**

****

* 1. **Presentation of the results**
  2. **Analysis and discussions**

**append(data)**

**Operation:** Adds a new node to the end of the list, maintaining the circular structure. This involves updating the next and prev pointers for the new node and the current tail node.

**Time Complexity**: O(1)

**insert\_after\_value(data\_after, data\_to\_insert)**

**Operation:** Searches for the first occurrence of data\_after in the list, then inserts a new node with data\_to\_insert after the found node.

**Time Complexity:** O(n)

**remove\_by\_value(data)**

**Operation:** Searches for the first occurrence of data in the list, then removes the corresponding node and updates the next and prev pointers of adjacent nodes.

**Time Complexity**: O(n)

**print\_forward()**

**Operation:** Traverses the list starting from the head node, collecting data from all nodes, and prints them**.**

**Time Complexity:** O(n)

**print\_backward()**

**Operation:** Traverses the list starting from the last node (self.head.prev), collecting data from all nodes in reverse order, and prints them.

**Time Complexity:** O(n)

**insert\_values(data\_list)**

**Operation**: Inserts multiple values into the list by calling the append method for each value**.**

**Time Complexity:** O(m)

|  |  |
| --- | --- |
| **Operation** | **Time Complexity** |
| append(data) | O(1) |
| insert\_after\_value() | O(n) |
| remove\_by\_value() | O(n) |
| print\_forward() | O(n) |
| print\_backward() | O(n) |
| insert\_values(data\_list) | O(n) |

# Experiment 3

**Title of the Laboratory Exercise**: Stack

1. **Aim:**

To understand and implement the basic operations in stack using python.

1. **Objective:**

To execute the below operations in stack:

1. Push: Pushing (storing) an element on the stack.
2. Pop: Removing (accessing) an element from the stack.
3. Peek: get the top data element of the stack, without removing it.
4. Check if stack is full.
5. Check if stack is empty.
6. **Exercise:**
7. Write a function in python that can reverse a string (your full name) using stack data structure. Create a function called “reverse\_myname” which does this operation.

Follow the steps given below to reverse a string using stack:

1. Create an empty stack.
2. One by one push all characters of string to stack by calling a push().
3. One by one pop all characters from stack and put them back to string
4. by calling a pop().
5. Create a Python function named "isit\_balanced" that determines if the string's paranthesis are balanced or not. "{}',"()" or "[]" are examples of parantheses.

**4. Experimental Procedure**

* 1. **Create your own list of your five-favourite sportsperson.** 
     1. **Algorithm design:**

class Stack:

push:

append(value)

pop:

if stack is not empty:

Begin

return stack.pop

End

else:

Begin

Error Pop from an empty stack

stack is empty:

len(self.stack) == 0

reverse a name:

create a Stack

for characters in the name

push the character

reversed name variable is created

while stack is not empty():

reversed name += popped element of stack

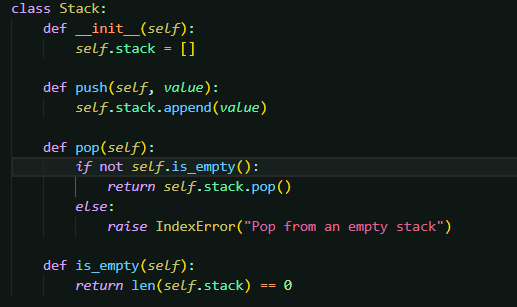
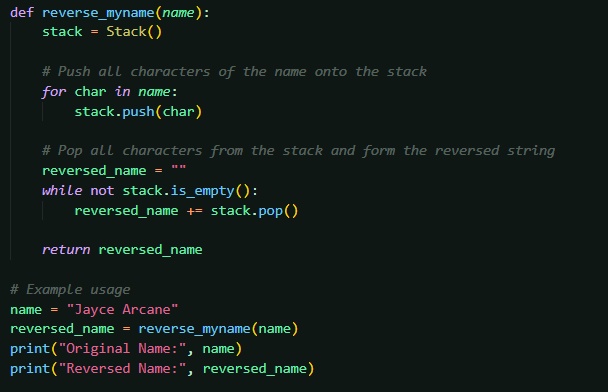
print reversed\_name

name that we want to reverse = "Jayce Arcane"

reversed name = reverse a name(variable name that we want to reverse)

print Original Name

print Reversed Name

* + 1. **Program:**
    2. **Presentation** **of the results:**
    3. **Analysis and discussions:**

**push(value):**

**Operation:** Appends a value to the end of the list (self.stack.append(value)).

**Time Complexity:** O(1)

**pop():**

**Operation:** Removes and returns the last value of the list (self.stack.pop()).

**Complexity:** O(1)

**is\_empty():**

**Operation:** Checks whether the stack is empty by comparing the length of the list to zero len == 0

**Time Complexity:** O(1)

**reverse\_myname(name):**

**Operation:** Reverses a string by pushing all characters to the stack and then popping them back in reverse order.

**Time Complexity:** O(n)

|  |  |
| --- | --- |
| **Operation** | **Time Complexity** |
| push(value) | O(1) |
| pop() | O(1) |
| is\_empty() | O(1) |
| reverse\_myname(name) | O(n) |

* 1. **Create a list of all even numbers between number x and number y.**
     1. **Algorithm design:**

class Stack:

init:

items = []

push:

items.append(item)

pop:

if is\_empty is false:

return items.pop()

return None

peek:

if is\_empty is false:

self.items[-1]

return None

is\_empty:

return len(items) == 0

size:

return len(items)

isit\_balanced:

create Stack()

matching\_pairs = {')': '(', '}': '{', ']': '['}

for char in string:

Begin

if char in ({[:

push(char)

elif char in )}]:

if

stack is\_empty or

stack pop() != matching\_pairs:

return False

End

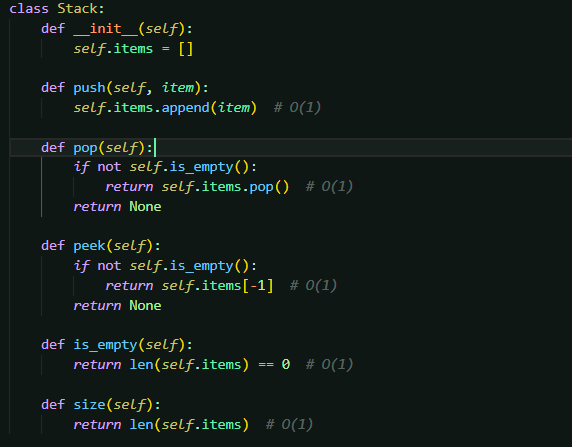
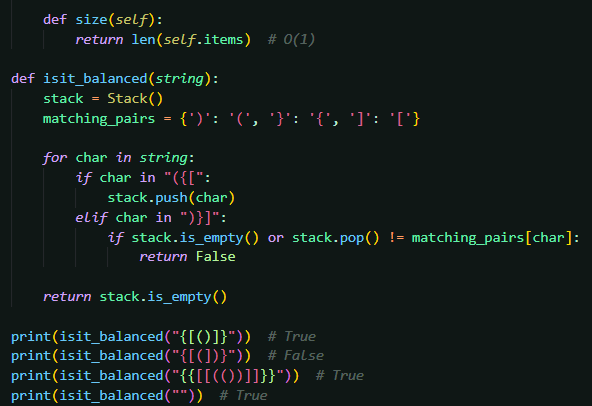
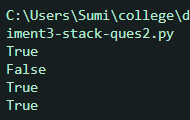
return stack.is\_empty()

print(is it balanced("{[()]}")) # True

print(is it balanced("{[(])}")) # False

print(is it balanced("{{[[(())]]}}")) # True

print(is it balanced("")) # True

* + 1. **Program:**
    2. **Presentation of the results:**
    3. **Analysis and discussions:**

**push(value):**

**Operation:** Adds a value to the stack by appending it to the end of the list (self.items.append(item)).

**Time Complexity:** O(1)

**pop():**

**Operation:** Removes and returns the last value in the stack using self.items.pop().

**Time Complexity:** O(1)

**peek():**

**Operation:** Returns the top value of the stack without removing it by accessing self.items[-1].

**Time Complexity:** O(1):

**is\_empty():**

**Operation:** Checks whether the stack is empty by comparing (len(self.items) == 0).

**Time Complexity**: O(1)

**isit\_balanced(string):**

**Operation:** Iterates over each character in the string, pushes all opening parentheses ((, {, [) to the stack, for closing parentheses (), }, ]), pops from the stack and checks for matching pairs, at the end, verifies if the stack is empty.

**Time Complexity:** O(n)

|  |  |
| --- | --- |
| **Operation** | **Time Complexity** |
| push(value) | O(1) |
| pop() | O(1) |
| peek() | O(1) |
| is\_empty() | O(1) |
| isit\_balanced() | O(n) |

# Experiment 4

**Title of the Laboratory Exercise**: Queue

1. **Aim:**

To understand and implement the basic operations in deque using python.

1. **Objective:**

To execute the below operations in a full binary tree:

Insert an element at the front end of the deque.

Delete an element at the rear end of the deque.

1. **Exercise:**

Using the deque data structure, insert some elements at the front and delete an element at the rear end of the deque. The maximum size of the array is 6. Check the conditions of overflow and underflow before carrying out insertion and deletion, respectively.

1. **Experimental Procedure**
   1. **Algorithm design**

class Deque

Begin

max\_size of deque = 6

queue = [None] \* max\_size

front element = -1

rear element = -1

is the deque full:

return (self.rear + 1) % self.max\_size == self.front

is the deque empty:

return self.front == -1

insert element at front:

if deque is full:

Begin

Overflow - Cannot insert.

End

if deque is empty:

Begin

self.front = self.rear = 0

End

else

Begin

self.front = (self.front - 1 + self.max\_size) % self.max\_size

End

self.queue[self.front] = value

delete element at the end:

if deque is empty:

Begin

Underflow - Cannot delete.

End

if self.front == self.rear:

Begin

self.front = self.rear = -1

End

else

Begin

self.rear = (self.rear - 1 + self.max\_size) % self.max\_size

End

display:

if self.is\_empty:

Begin

Deque is empty

End

print("Deque contents:")

while index = self.front

Begin

print self.queue[index]

if index == self.rear

Begin

break

index = (index + 1) % self.max\_size

End

print

End

create Deque

# Insert elements at the front

insert element at start(10)

insert element at start(20)

insert element at start(30)

insert element at start(40)

insert element at start(50)

insert element at start(60)

insert element at start(70)

display dequeue

delete element from the end

delete element from the end

display dequeue

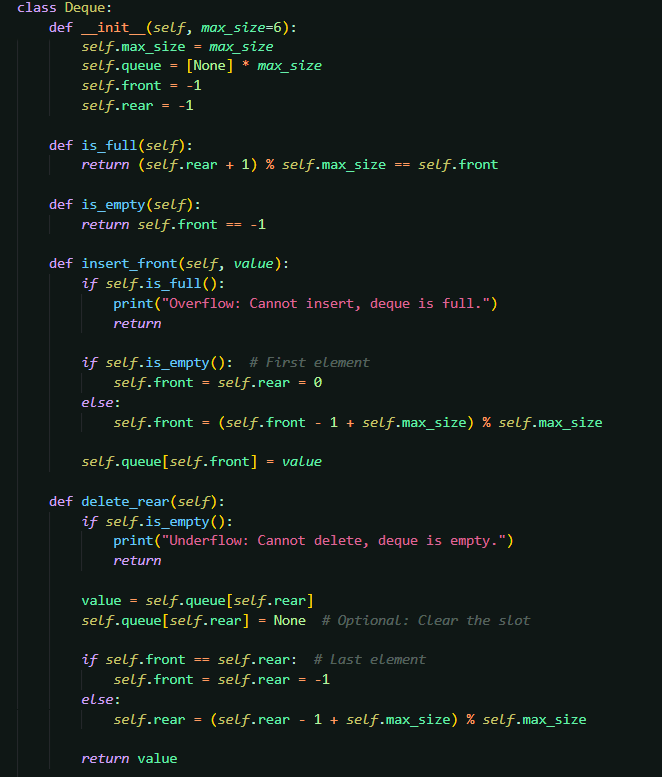
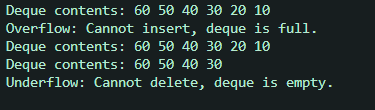
delete element from the end

delete element from the end

delete element from the end

delete element from the end

delete element from the end

* 1. ******Program**
  2. **Presentation of the results**
  3. **Analysis and discussions**

**is\_full: Operation:** Checks whether the deque is full by comparing (rear + 1) % max\_size to front.

**Time Complexity:** O(1)O(1)

**is\_empty: Operation:** Checks whether the deque is empty by checking if front equals -1.

**Time Complexity:** O(1)O(1)

**insert\_front: Operation:** Inserts an element at the front of the deque.

**Steps:**

* 1. Check if the deque is full using is\_full.
  2. Update the front index to the previous slot in a circular manner.
  3. Insert the value at the updated front position.

**Time Complexity:** O(1)O(1)

**delete\_rear: Operation:** Deletes an element from the rear of the deque.

**Steps:**

* 1. Check if the deque is empty using is\_empty.
  2. Retrieve the value at the current rear.
  3. Update the rear index to the previous slot in a circular manner.

**Time Complexity:** O(1)O(1)

**Display: Operation:** Displays all elements in the deque in order from front to rear.

**Steps:**

* 1. Start from the front index.
  2. Traverse the deque circularly until reaching the rear.

**Time Complexity:** O(n)O(n)

**Summary of Time Complexities**

**Efficiency**

|  |  |
| --- | --- |
| **Operation** | **Time Complexity** |
| is\_full | O(1) |
| is\_empty | O(1) |
| insert\_front | O(1) |
| delete\_rear | O(1) |
| display | O(n) |