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	

Experiment 1

Title of the laboratory experiment: Array

1. **Aim:**
To understand and implement the basic operations in arrays using python.

2. **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.

3. **Exercise:**
To develop a python to perform the below tasks:
    1. Create your own list of your favourite five sportsperson. Using this find out,
        a) Length of the list.
        b) Add a sixth sportsperson at the end of this list.
        c) 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.
        d) Now you don't like two sportspersons. Now remove those two and replace them with any other two sportspersons.
        e) Sort the sportspersons list in alphabetical order (hint: use the dir() functions to list down all functions available in the list).
    2. 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.

4. **Experimental Procedure**

4.1. Create your own list of your five-favourite sportsperson.

4.1.1. 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()
```

4.1.2. Program:

```python
class SportspersonArray:
    def __init__(self, initial_list):
        self.sportspersons = initial_list

    def get_length(self):
        return len(self.sportspersons)  # O(1)

    def add_at_end(self, sportsperson):
        self.sportspersons.append(sportsperson)  # O(1)

    def add_after_second(self, sportsperson):
        if len(self.sportspersons) < 2:
            print("Not enough elements to add after the second position.")
            return
        # Remove the sportsperson first
        self.sportspersons.remove(sportsperson)  # O(n)
        # Insert it after the second sportsperson
        self.sportspersons.insert(2, sportsperson)  # O(n)

    def replace_two_sportspersons(self, remove_list, new_list):
        for person in remove_list:
            if person in self.sportspersons:
                self.sportspersons.remove(person)  # O(n)
        self.sportspersons.extend(new_list)  # O(m), where m = len(new_list)

    def sort_sportspersons(self):
        self.sportspersons.sort()  # O(n log n)

    def print_sportspersons(self):
        print("Sportspersons:", self.sportspersons)

# Create the array with your favorite five sportspersons
favourite_sportspersons = SportspersonArray(["Hamilton",
                         "Vettel", "Leclerc", "Sainz", "Ricciardo"])

# Task 1(a): Length of the list
print("Length of list:", favourite_sportspersons.get_length())

# Task 1(b): Add a sixth sportsperson at the end
favourite_sportspersons.add_at_end("Yuki")
favourite_sportspersons.print_sportspersons()

# Task 1(c): Add the sixth sportsperson after the second sportsperson
favourite_sportspersons.add_after_second("Yuki")
favourite_sportspersons.print_sportspersons()

# Task 1(d): Remove two sportspersons and replace them with two others
favourite_sportspersons.replace_two_sportspersons(["Ricciardo",
                              "Leclerc"], ["Schumacher", "Senna"])
favourite_sportspersons.print_sportspersons()

# Task 1(e): Sort the list alphabetically
favourite_sportspersons.sort_sportspersons()
favourite_sportspersons.print_sportspersons()
```

4.1.3. Presentation of the results:

```
C:\Users\Sumi\college\dsa\documentation>C:/Python312/python.exe c:/Users/Sumi/college/
ray-ques1.py
Length of list: 5
Sportspersons: ['Hamilton', 'Vettel', 'Leclerc', 'Sainz', 'Ricciardo', 'Yuki']
Sportspersons: ['Hamilton', 'Vettel', 'Yuki', 'Leclerc', 'Sainz', 'Ricciardo']
Sportspersons: ['Hamilton', 'Vettel', 'Yuki', 'Sainz', 'Schumacher', 'Senna']
Sportspersons: ['Hamilton', 'Sainz', 'Schumacher', 'Senna', 'Vettel', 'Yuki']
```

4.1.4. 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)

4.2. Create a list of all even numbers between number x and number y.

4.2.1. 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()
```

4.2.2. Program:

```python
class EvenNumbersArray:
    def __init__(self):
        self.even_numbers = []

    def generate_even_numbers(self, x, y):
        # Ensure that x is less than y
        if x > y:
            print("Invalid range: x should be less than y.")
            return
        # Add even numbers to the array
        for num in range(x, y + 1):
            if num % 2 == 0:
                self.even_numbers.append(num)  # O(1) per append

    def print_even_numbers(self):
        print("Even Numbers:", self.even_numbers)  # O(n)
```

```python
    def get_length(self):
        return len(self.even_numbers)  # O(1)

# Create the EvenNumbersArray instance
x = 19  # Your age
y = 51  # Father's age
even_numbers_array = EvenNumbersArray()

# Generate even numbers between x and y
even_numbers_array.generate_even_numbers(x, y)

# Print the even numbers and the length of the array
even_numbers_array.print_even_numbers()
print("Length of the list:", even_numbers_array.get_length())
```

4.2.3. Presentation of the results:

```
C:\Users\Sumi\college\dsa\documentation>C:/Python312/python.exe c:/Users/Sumi/colle
ray-ques2.py
Even Numbers: [20, 22, 24, 26, 28, 30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50]
Length of the list: 16

C:\Users\Sumi\college\dsa\documentation>
```

4.2.4. 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.

2. **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.

3. **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

4.1. 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()
```

4.2. Program

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

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

    def insert_values(self, data_list):
        for data in data_list:
            self.append(data)

    def append(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            self.head.next = self.head
            self.head.prev = self.head
        else:
            tail = self.head.prev
            tail.next = new_node
            new_node.prev = tail
            new_node.next = self.head
            self.head.prev = new_node

    def insert_after_value(self, data_after, data_to_insert):
        if not self.head:
            print(f"List is empty. Cannot insert {data_to_insert} after {data_after}.")
            return

        current = self.head
        while True:
            if current.data == data_after:
                new_node = Node(data_to_insert)
                new_node.next = current.next
                new_node.prev = current
                current.next.prev = new_node
                current.next = new_node
                return
            current = current.next
            if current == self.head:
                break
        print(f"Value {data_after} not found in the list.")

    def remove_by_value(self, data):
        if not self.head:
            print(f"List is empty. Cannot remove {data}.")
            return

        current = self.head
        while True:
            if current.data == data:
                if current.next == current:
                    self.head = None
                else:
                    current.prev.next = current.next
```

```python
class CircularDoublyLinkedList:
    def remove_by_value(self, data):
                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(f"Value {data} not found in the list.")

    def print_forward(self):
        if not self.head:
            print("List is empty.")
            return

        current = self.head
        result = []
        while True:
            result.append(current.data)
            current = current.next
            if current == self.head:
                break
        print(" -> ".join(result))

    def print_backward(self):
        if not self.head:
            print("List is empty.")
            return

        current = self.head.prev
        result = []
        while True:
            result.append(current.data)
            current = current.prev
            if current == self.head.prev:
                break
        print(" <- ".join(result))


# Testing the Circular Doubly Linked List
LL = CircularDoublyLinkedList()
LL.insert_values(["Red", "Yellow", "Purple", "Orange"])
LL.print_forward()
LL.insert_after_value("Yellow", "Blue")
LL.print_forward()
LL.remove_by_value("Orange")
LL.print_forward()
LL.remove_by_value("Green")
LL.print_forward()
LL.remove_by_value("Red")
LL.remove_by_value("Yellow")
LL.remove_by_value("Blue")
LL.remove_by_value("Purple")
LL.print_forward()
LL.print_forward()
LL.print_backward()
```

4.3. Presentation of the results

```
C:\Users\Sumi\college\dsa\documentation>C:/Python312/python.exe c:/Users/Sumi/
nkedlist.py
Red -> Yellow -> Purple -> Orange
Red -> Yellow -> Blue -> Purple -> Orange
Red -> Yellow -> Blue -> Purple
Value Green not found in the list.
Red -> Yellow -> Blue -> Purple
List is empty.
List is empty.
List is empty.
```

4.4. 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.

2. **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.

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

<u>4. Experimental Procedure</u>

4.1. Create your own list of your five-favourite sportsperson.

4.1.1. 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

4.1.2. Program:

```
class Stack:
    def __init__(self):
        self.stack = []

    def push(self, value):
        self.stack.append(value)

    def pop(self):
        if not self.is_empty():
            return self.stack.pop()
        else:
            raise IndexError("Pop from an empty stack")

    def is_empty(self):
        return len(self.stack) == 0
```

```
def reverse_myname(name):
    stack = Stack()

    # Push all characters of the name onto the stack
    for char in name:
        stack.push(char)

    # Pop all characters from the stack and form the reversed string
    reversed_name = ""
    while stack.is_empty():
        reversed_name += stack.pop()

    return reversed_name

# Example usage
name = "Jayce Arcane"
reversed_name = reverse_myname(name)
print("Original Name:", name)
print("Reversed Name:", reversed_name)
```

4.1.3. Presentation of the results:

```
Original Name: Jayce Arcane
Reversed Name: enacrA ecyaJ
```

4.1.4. 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)

4.2. Create a list of all even numbers between number x and number y.

4.2.1. 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

4.2.2. Program:

```python
class Stack:
    def __init__(self):
        self.items = []

    def push(self, item):
        self.items.append(item)  # O(1)

    def pop(self):
        if not self.is_empty():
            return self.items.pop()  # O(1)
        return None

    def peek(self):
        if not self.is_empty():
            return self.items[-1]  # O(1)
        return None

    def is_empty(self):
        return len(self.items) == 0  # O(1)

    def size(self):
        return len(self.items)  # O(1)
```

```python
    def size(self):
        return len(self.items)  # O(1)

def isit_balanced(string):
    stack = Stack()
    matching_pairs = {')': '(', '}': '{', ']': '['}

    for char in string:
        if char in "({[":
            stack.push(char)
        elif char in ")}]":
            if stack.is_empty() or stack.pop() != matching_pairs[char]:
                return False

    return stack.is_empty()

print(isit_balanced("{[()]}"))  # True
print(isit_balanced("{[(])}"))  # False
print(isit_balanced("{{[[(())]]}}"))  # True
print(isit_balanced(""))  # True
```

4.2.3. Presentation of the results:

```
C:\Users\Sumi\college\d
iment3-stack-ques2.py
True
False
True
True
```

4.2.4. 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.

2. **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.

3. **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.

4. **Experimental Procedure**

4.1. 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

4.2. Program

```python
class Deque:
    def __init__(self, max_size=6):
        self.max_size = max_size
        self.queue = [None] * max_size
        self.front = -1
        self.rear = -1

    def is_full(self):
        return (self.rear + 1) % self.max_size == self.front

    def is_empty(self):
        return self.front == -1

    def insert_front(self, value):
        if self.is_full():
            print("Overflow: Cannot insert, deque is full.")
            return

        if self.is_empty():  # First element
            self.front = self.rear = 0
        else:
            self.front = (self.front - 1 + self.max_size) % self.max_size

        self.queue[self.front] = value

    def delete_rear(self):
        if self.is_empty():
            print("Underflow: Cannot delete, deque is empty.")
            return

        value = self.queue[self.rear]
        self.queue[self.rear] = None  # Optional: Clear the slot

        if self.front == self.rear:  # Last element
            self.front = self.rear = -1
        else:
            self.rear = (self.rear - 1 + self.max_size) % self.max_size

        return value
```

```python
    def display(self):
        if self.is_empty():
            print("Deque is empty.")
            return

        print("Deque contents:", end=" ")
        index = self.front
        while True:
            print(self.queue[index], end=" ")
            if index == self.rear:
                break
            index = (index + 1) % self.max_size
        print()

# Example usage
deque = Deque()

deque.insert_front(10)
deque.insert_front(20)
deque.insert_front(30)
deque.insert_front(40)
deque.insert_front(50)
deque.insert_front(60)

deque.display()

deque.insert_front(70)

deque.display()

deque.delete_rear()
deque.delete_rear()

deque.display()

deque.delete_rear()
deque.delete_rear()
deque.delete_rear()
deque.delete_rear()  # Deleting until empty

deque.delete_rear()
```

4.3. Presentation of the results

```
Deque contents: 60 50 40 30 20 10
Overflow: Cannot insert, deque is full.
Deque contents: 60 50 40 30 20 10
Deque contents: 60 50 40 30
Underflow: Cannot delete, deque is empty.
```

4.4. 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)