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)
```

```python
    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:

```python
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
```

```python
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

    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)

Experiment 5

Title of the Laboratory Exercise: Binary Tree

1. **Aim**:

   To understand and implement the basic operations in full binary tree using python.

2. **Objective**:

   To execute the below operations in a full binary tree:

   1. Search − Searches an element in a tree.
   2. Insert − Inserts an element in a tree.
   3. Pre-order Traversal − Traverses a tree in a pre-order manner.
   4. In-order Traversal − Traverses a tree in an in-order manner.
   5. Post-order Traversal − Traverses a tree in a post-order manner.

3. **Exercise:**

   Construct a full binary tree with 10 nodes, where the data item inserted at every node should be a random value between 1 and 100. Add the following methods to the class named "FullBinaryTree" and perform the operation on the constructed full binary tree.

   1. find_min(): finds the minimum element stored in the constructed Full binary tree.
   2. find_max(): finds the maximum element stored in the constructed Full binary tree.
   3. calculate_sum(): calculates the sum of all elements stored in the constructed Full binary tree.
   4. pre_order_traversal(): performs pre-order traversal of the constructed Full binary tree.
   5. post_order_traversal(): performs post-order traversal of the constructed Full binary tree.
   6. in_order_traversal(): performs in-order traversal of the constructed Full binary tree.

4. **Experimental Procedure**

   4.1 Algorithm design

```
import random

class Node:
  def init:
    data, left, right = None

class FullBinaryTree:
  def init:
    self.root = None

  def construct_full_binary_tree(self, n=10, min_value=1, max_value=100):
    values = random.randint(min_value, max_value)
    self.root = constructtree(values, 0)

  def _construct_tree(self, values, index):
    if index < len(values):
      node.left = constructtree(values, 2 * index + 1)
      node.right = constructtree(values, 2 * index + 2)
      return node
    return None
```

```
  def find_min:
    return min(data, findmin(node.left), findmin(node.right))

  def findmax:
    return max(data, findmax(node.left), findmax(node.right))

  def calculatesum:
    if not node:
      return 0
    return data + calculatesum(node.left) + calculatesum(node.right)

  def pre_order_traversal(self, node):
    stack = []
    current = node
    while stack or current:
      if current:
        stack.append(current)
        current = current.left
      else:
        current = stack.pop()
        current = current.right

  def post_order_traversal(self, node):
    stack1 = []
    stack2 = []
    if node:
      stack1.append(node)
    while stack1:
      current = stack1.pop()
      stack2.append(current)
      if current.left:
        append(current.left)
      if current.right:
        append(current.right)


  def inordertraversal:
    stack = []
    current = node
    while stack or current:
      if current:
        stack.append(current)
        current = current.left
      else:
        current = stack.pop()
        current = current.right
main:
```

fbt = FullBinaryTree()

constructfullbinarytree()

inordertraversal(root)
preordertraversal(root)
postordertraversal(root)
print(findmin(root))
print(findmax(root))
print(calculate_sum(root))

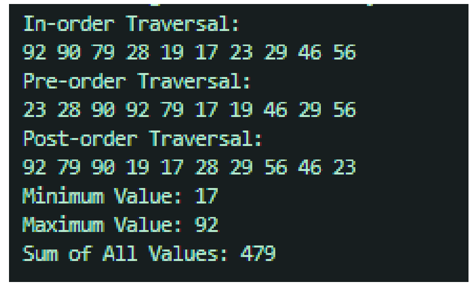
4.2 Program

```python
import random

class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

class FullBinaryTree:
    def __init__(self):
        self.root = None

    # Construct a full binary tree with random values
    def construct_full_binary_tree(self, n=10, min_value=1, max_value=100):
        """Constructs a full binary tree with n nodes."""
        if n <= 0:
            return
        values = [random.randint(min_value, max_value) for _ in range(n)]
        self.root = self._construct_tree(values, 0)

    def _construct_tree(self, values, index):
        """Helper function to recursively construct a full binary tree."""
        if index < len(values):
            node = Node(values[index])
            node.left = self._construct_tree(values, 2 * index + 1)
            node.right = self._construct_tree(values, 2 * index + 2)
            return node
        return None

    # Find the minimum value in the tree
    def find_min(self, node):
        if not node:
            return float('inf')
        return min(node.data, self.find_min(node.left), self.find_min(node.right))

    # Find the maximum value in the tree
    def find_max(self, node):
        if not node:
            return float('-inf')
        return max(node.data, self.find_max(node.left), self.find_max(node.right))

```

```python
class FullBinaryTree:
    def calculate_sum(self, node):
            return 0
        return node.data + self.calculate_sum(node.left) + self.calculate_sum(node.right)

    # Pre-order Traversal: Root -> Left -> Right
    def pre_order_traversal(self, node):
        stack = []
        current = node
        while stack or current:
            if current:
                print(current.data, end=" ")
                stack.append(current)
                current = current.left
            else:
                current = stack.pop()
                current = current.right

    # Post-order Traversal: Left -> Right -> Root
    def post_order_traversal(self, node):
        stack1 = []
        stack2 = []
        if node:
            stack1.append(node)
        while stack1:
            current = stack1.pop()
            stack2.append(current)
            if current.left:
                stack1.append(current.left)
            if current.right:
                stack1.append(current.right)
        while stack2:
            print(stack2.pop().data, end=" ")

    # In-order Traversal: Left -> Root -> Right
    def in_order_traversal(self, node):
        stack = []
        current = node
        while stack or current:
            if current:
                stack.append(current)
                current = current.left
            else:
                current = stack.pop()
                print(current.data, end=" ")
                current = current.right
```

```python
# Example Usage
if __name__ == "__main__":
    # Create a FullBinaryTree instance
    fbt = FullBinaryTree()

    # Construct the tree with random values
    fbt.construct_full_binary_tree()

    # Perform operations
    print("In-order Traversal:")
    fbt.in_order_traversal(fbt.root)
    print("\nPre-order Traversal:")
    fbt.pre_order_traversal(fbt.root)
    print("\nPost-order Traversal:")
    fbt.post_order_traversal(fbt.root)
    print("\nMinimum Value:", fbt.find_min(fbt.root))
    print("Maximum Value:", fbt.find_max(fbt.root))
    print("Sum of All Values:", fbt.calculate_sum(fbt.root))
```

4.3 Presentation of the results



4.4 Analysis and discussions

Insert (insert and _insert):
Operation: Adds a new node to the tree, ensuring the tree remains full.
Time Complexity: O(n) (skewed tree)

Pre-order Traversal:
Operation: Visits nodes in the order: root, left subtree, right subtree
Time Complexity: O(n)

Post-order Traversal:
Operation: Visits nodes in the order: left subtree, right subtree, root
Time Complexity: O(n)

Level-order Traversal:
Operation: Visits nodes level by level from top to bottom and left to right
Time Complexity: O(n)

Branch-wise Traversal:
Operation: Similar to level-order but focuses on nodes at each branch level.
Time Complexity: O(n)

Find Minimum (find_min):
Operation: Recursively finds the minimum value by comparing the node values
Time Complexity: O(n)

Find Maximum (find_max):
Operation: Recursively finds the maximum value by comparing the node values.
Time Complexity: O(n)

FUNCTION	**Time Complexity**:
Pre-order Traversal:	O(n)
Post-order Traversal:	O(n)
Level-order Traversal:	O(n)
Branch-wise Traversal:	O(n)
Find Minimum (find_min):	O(n)
Find Maximum (find_max):	O(n)
Insert (insert and _insert):	O(n)

Experiment 6

Title of the Laboratory Exercise: Binary Search Tree

1. **Aim**:
   To understand and implement the basic operations in Binary Search Tree using python.

2. **Objective**:
   To execute the below operations in a Binary Search Tree (BST):
   1. Search − Searches an element in a BST.
   2. Insert − Inserts an element in a BST.
   3. Delete − Deletes an element in a BST.
   4. Check the balance of the BST.
   5. Determine the height of the BST.

3. **Exercise**:
   Construct a binary search tree with the below values: {12, 35, 14, 97, 36, 65, 89}. Write a python program to perform the following operations:
   1. Insert a new element which is having a value equivalent to the "last two digits of your roll number".
   2. To determine the height of the constructed BST.
   3. Delete any element from the constructed BST.
   4. To check if the constructed BST is Balanced or not.

4. **Experimental Procedure**

   4.1 Algorithm design

```
class Node:
   def init:
      data, left, right


class BST:
   def init:
      root = None

   def insert(self, data):
      if root is None:
         root = Node(data)
      else:
         insert(data)

   def insert:
      if data < current.data:
         if left is None:
            left = Node(data)
         else:
            insert(data)
      else:
```

```
        if right is None:
            right = Node(data)
        else:
            insert(data)

def height:
    if not node:
        return -1
    leftheight = height(node.left)
    rightheight = height(node.right)
    return 1 + max(leftheight, rightheight)

def delete:
    root = delete(data)

def delete:
    if data < node.data:
        node.left = delete(data)
    elif data > node.data:
        node.right = delete(data)
    else:
       not node.right:
            return node.left
        temp = minvaluenode(node.right)
        node.data = temp.data
        node.right = delete(temp.data)
    return node

def minvaluenode:
    current = node
    while current.left:
        current = current.left
    return current

def is_balanced:
    if not node:
        return True
    left_height = height(node.left)
    right_height = height(node.right)
    if abs(left_height - right_height) > 1:
        return False
    return isbalanced(node.left) and isbalanced(node.right)

def inorder:
    if node:
        inorder(node.left)
        print(node.data, end=" ")
        self.in_order(node.right)
```

```
bst = BST()

values = [12, 35, 14, 97, 36, 65, 89]
for value in values:
   bst.insert(value)
roll_number_last_two_digits = 12
bst.insert(roll_number_last_two_digits)


tree_height = bst.height(bst.root)
print(f"Height of the BST: {tree_height}")


bst.delete(14)
bst.in_order(bst.root)
is_bal = bst.is_balanced(bst.root)
```

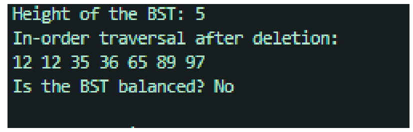
4.2 Program

```python
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None


class BST:
    def __init__(self):
        self.root = None

    # Insert a new element
    def insert(self, data):
        if self.root is None:
            self.root = Node(data)
        else:
            self._insert(self.root, data)

    def _insert(self, current, data):
        if data < current.data:
            if current.left is None:
                current.left = Node(data)
            else:
                self._insert(current.left, data)
        else:
            if current.right is None:
                current.right = Node(data)
            else:
                self._insert(current.right, data)

    # Determine the height of the tree
    def height(self, node):
        if not node:
            return -1  # Height of an empty tree is -1
        left_height = self.height(node.left)
        right_height = self.height(node.right)
        return 1 + max(left_height, right_height)

    # Delete a node
    def delete(self, data):
        self.root = self._delete(self.root, data)
```

```python
    # Delete a node
    def delete(self, data):
        self.root = self._delete(self.root, data)

    def _delete(self, node, data):
        if not node:
            return node
        if data < node.data:
            node.left = self._delete(node.left, data)
        elif data > node.data:
            node.right = self._delete(node.right, data)
        else:
            # Node with one or no child
            if not node.left:
                return node.right
            elif not node.right:
                return node.left
            # Node with two children: Get inorder successor
            temp = self._min_value_node(node.right)
            node.data = temp.data
            node.right = self._delete(node.right, temp.data)
        return node

    def _min_value_node(self, node):
        current = node
        while current.left:
            current = current.left
        return current

    # Check if the tree is balanced
    def is_balanced(self, node):
        if not node:
            return True
        left_height = self.height(node.left)
        right_height = self.height(node.right)
        if abs(left_height - right_height) > 1:
            return False
        return self.is_balanced(node.left) and self.is_balanced(node.right)

    # In-order traversal (helper to visualize the tree)
    def in_order(self, node):
        if node:
            self.in_order(node.left)
            print(node.data, end=" ")
            self.in_order(node.right)

```

```python
# Example usage
bst = BST()

# Construct BST with given values
values = [12, 35, 14, 97, 36, 65, 89]
for value in values:
    bst.insert(value)

# 1. Insert new element
roll_number_last_two_digits = 12  # Replace with your own roll number's last two digits
bst.insert(roll_number_last_two_digits)

# 2. Determine height
tree_height = bst.height(bst.root)
print(f"Height of the BST: {tree_height}")

# 3. Delete an element
bst.delete(14)  # Example: Deleting 14
print("In-order traversal after deletion:")
bst.in_order(bst.root)

# 4. Check if the tree is balanced
is_bal = bst.is_balanced(bst.root)
print(f"\nIs the BST balanced? {'Yes' if is_bal else 'No'}")

```

4.3 Presentation of the results



4.4 Analysis and discussions

Insert a New Element:

Operation: Adds a new node to the BST while maintaining the BST property (left child < parent node < right child).

Time Complexity: O(log n)

Determine the Height of BST:

Operation: Calculates the height of the BST, which is the number of edges on the longest path from the root to a leaf**.**

Time Complexity: O(n)

Delete an Element:

Operation: Removes a node from the BST while maintaining the BST property. Depending on the node to be deleted (leaf, one child, two children), different cases need to be handled.

Time Complexity: O(n^2

In-order Traversal:

Operation: Visits nodes in the order: left subtree, root, right subtree.

Time Complexity: O(n)

FUNCTION	**Time Complexity**:
Insert a New Element:	O(log n)
Determine the Height of BST:	O(n)
Delete an Element:	O(n^2
In-order Traversal:	O(n)

Experiment 7

Title of the Laboratory Exercise: Heap

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

2. **Objective**:
   To execute the below operations in a Heap: https://medium.com/techie-delight/heap-practice-problems-and-interview-questions-b678ff3b694c

3. **Exercise**:
   10, 12, 14, 16, 18 and 20 and perform the following operation on the constructed Heap Tree.
   1. Insert a new element whose value is equivalent to the sum of the digits of your roll number.
   2. Find the maximum element in the constructed Max Heap.
   3. Delete the root element (maximum element) two times from the Max Heap.

4. **Experimental Procedure**

   4.1 Algorithm design

```
import heapq

class MaxHeap:
  init:
    self.heap = []

  def buildheap:
    heap = [-el for el in elements]
    heapify(heap)

  def insert:
    heappush(self.heap, -value)

  def findmax:
    return -self.heap[0] if self.heap else None

  def delete_max(self):
    return -heapq.heappop(self.heap) if self.heap else None

  def print_heap:
    print([-el for el in self.heap])

__main__:
  elements = [10, 12, 14, 16, 18, 20]
  heap = MaxHeap()
  heap.buildheap(elements)

  print("Initial Max Heap:")
  heap.print_heap()
```

roll_number = 412012
sum_of_digits = sum(digit)
heap.insert(sum_of_digits)
heap.print_heap()

max_element = heap.find_max()
print(max_element)

deleted_1 = heap.delete_max()
print(deleted_1)
heap.print_heap()
deleted_2 = heap.delete_max()
print(deleted_2)
heap.print_heap()

4.2 Program

```python
import heapq

class MaxHeap:
    def __init__(self):
        self.heap = []

    # Convert a list into a Max Heap
    def build_heap(self, elements):
        self.heap = [-el for el in elements]  # Negate to use min-hea
        heapq.heapify(self.heap)

    # Insert a new element
    def insert(self, value):
        heapq.heappush(self.heap, -value)

    # Find the maximum element
    def find_max(self):
        return -self.heap[0] if self.heap else None

```

```python

    # Delete the maximum element (root)
    def delete_max(self):
        return -heapq.heappop(self.heap) if self.heap else None

    # Print the heap
    def print_heap(self):
        print([-el for el in self.heap])

# Example Usage
if __name__ == "__main__":
    elements = [10, 12, 14, 16, 18, 20]
    heap = MaxHeap()
    heap.build_heap(elements)

    print("Initial Max Heap:")
    heap.print_heap()

    # 1. Insert a new element (sum of digits of roll number)
    roll_number = 412012 # Replace with your roll number
    sum_of_digits = sum(int(digit) for digit in str(roll_number))
    print(f"\nInserting element (sum of digits of roll number): {sum_of_digits}")
    heap.insert(sum_of_digits)
    print("Heap after insertion:")
    heap.print_heap()

    # 2. Find the maximum element
    max_element = heap.find_max()
    print(f"\nMaximum element in the Max Heap: {max_element}")

    # 3. Delete the root element (maximum element) twice
    print("\nDeleting the root element (maximum) twice:")
    deleted_1 = heap.delete_max()
    print(f"Deleted element: {deleted_1}")
    heap.print_heap()
    deleted_2 = heap.delete_max()
    print(f"Deleted element: {deleted_2}")
    heap.print_heap()
```

4.3 Presentation of the results

```
Initial Max Heap:
[20, 18, 14, 16, 12, 10]

Inserting element (sum of digits of roll number): 10
Heap after insertion:
[20, 18, 14, 16, 12, 10, 10]

Maximum element in the Max Heap: 20

Deleting the root element (maximum) twice:
Deleted element: 20
[18, 16, 14, 10, 12, 10]
Deleted element: 18
[16, 12, 14, 10, 10]
```

4.4 Analysis and discussions

Heap Construction:

Operation: Converts an unsorted list of elements into a Max Heap. This is done by negating the elements to leverage Python's heapq (which is a Min Heap) to simulate a Max Heap.

Time Complexity: O(n)

Insertion:

Operation: Inserts a new element into the Max Heap while maintaining the heap property. The element is added, and then the heap is restructured (up-heap) to ensure the max-heap property is upheld

Time Complexity: O(log n)

Find Maximum:

Operation: Retrieves the maximum element in the Max Heap, which is always at the root (index 0).

Time Complexity: O(1)

Delete Maximum:

Operation: Deletes the maximum element (root) from the Max Heap. The last element is moved to the root, and then the heap is restructured (down-heap) to maintain the max-heap property.

Time Complexity: O(log n)

Experiment 8

Title of the Laboratory Exercise: AVL Tree

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

2. **Objective:**
   To execute the below operations in an AVL Tree:
   1. Left rotation
   2. Right rotation
   3. Left-Right rotation
   4. Right-Left rotation

3. **Exercise:**
   Implement a Python program that constructs an AVL tree having the following elements: Z, I, J, F, A, E, C, P, B, D, H, N. Consider the order of the elements in ascending order. Explain the rotations diagrammatically.

4. **Experimental Procedure**

   4.1 Algorithm design

```
class Node:
  def init:
    self.data, left, right, height = 1

class AVLTree:
  def init:
    root = None

  def height:
    return node.height if node else 0

  def getbalance:
    return height(node.left) - height(node.right)

  def rightrotate:
    y = z.left
    T3 = y.right

    y.right = z
    z.left = T3

    z.height = 1 + max(height(z.left), height(z.right))
    y.height = 1 + max(height(y.left), height(y.right))

    return y

  def leftrotate:
    y = z.right
```

```
        T2 = y.left

        y.left = z
        z.right = T2

        z.height = 1 + max(height(z.left), height(z.right))
        y.height = 1 + max(height(y.left), height(y.right))

        return y

    def insert:
        if data < root.data:
            left = insert(left, data)
        elif data > root.data:
            right = insert(right, data)
        else:
            return root

        root.height = 1 + max(height(root.left), height(root.right))

        balance = getbalance(root)

        if balance > 1 and data < left.data:
            return rightrotate(root)

        if balance < -1 and data > root.right.data:
            return eftrotate(root)

        if balance > 1 and data > root.left.data:
            root.left = leftrotate(root.left)
            return rightrotate(root)

        if balance < -1 and data < root.right.data:
            root.right = rightrotate(root.right)
            return leftrotate(root)
        return root

    def inorder:
        if root:
            inorder(root.left)
            print(root.data)
            inorder(root.right)

__main__:
    elements = ["Z", "I", "J", "F", "A", "E", "C", "P", "B", "D", "H", "N"]
    elements.sort()

    avl = AVLTree()
```

root = None

for element in elements:
    root = avl.insert(root, element)

avl.in_order(root)

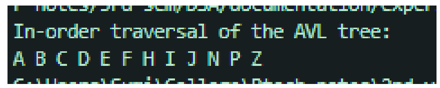
4.2 Program

```python
class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None
        self.height = 1  # Height of the node


class AVLTree:
    def __init__(self):
        self.root = None

    # Get the height of a node
    def _height(self, node):
        return node.height if node else 0

    # Get the balance factor of a node
    def _get_balance(self, node):
        if not node:
            return 0
        return self._height(node.left) - self._height(node.right)

    # Right rotate
    def _right_rotate(self, z):
        y = z.left
        T3 = y.right

        # Perform rotation
        y.right = z
        z.left = T3

        # Update heights
        z.height = 1 + max(self._height(z.left), self._height(z.right))
        y.height = 1 + max(self._height(y.left), self._height(y.right))

        # Return new root
        return y

    # Left rotate
    def _left_rotate(self, z):
        y = z.right
        T2 = y.left

        # Perform rotation
        y.left = z
        z.right = T2

        # Update heights
        z.height = 1 + max(self._height(z.left), self._height(z.right))
        y.height = 1 + max(self._height(y.left), self._height(y.right))

        # Return new root
        return y

    # Insert a node
    def insert(self, root, data):
```

```python
    # Insert a node
    def insert(self, root, data):
        if not root:
            return Node(data)

        if data < root.data:
            root.left = self.insert(root.left, data)
        elif data > root.data:
            root.right = self.insert(root.right, data)
        else:
            return root  # Duplicate keys not allowed

        # Update height of this node
        root.height = 1 + max(self._height(root.left), self._height(root.right))

        # Get balance factor to check if the node is unbalanced
        balance = self._get_balance(root)

        # Balance the node using rotations
        # Case 1: Left Left
        if balance > 1 and data < root.left.data:
            return self._right_rotate(root)

        # Case 2: Right Right
        if balance < -1 and data > root.right.data:
            return self._left_rotate(root)

        # Case 3: Left Right
        if balance > 1 and data > root.left.data:
            root.left = self._left_rotate(root.left)
            return self._right_rotate(root)

        # Case 4: Right Left
        if balance < -1 and data < root.right.data:
            root.right = self._right_rotate(root.right)
            return self._left_rotate(root)

        return root

    # In-order traversal
    def in_order(self, root):
        if root:
            self.in_order(root.left)
            print(root.data, end=" ")
            self.in_order(root.right)

# Example usage
if __name__ == "__main__":
    elements = ["Z", "I", "J", "F", "A", "E", "C", "P", "B", "D", "H", "N"]
    elements.sort()  # Sort elements in ascending order

    avl = AVLTree()
    root = None
```

```python
# Example usage
if __name__ == "__main__":
    elements = ["Z", "I", "J", "F", "A", "E", "C", "P", "B", "D", "H", "N"]
    elements.sort()  # Sort elements in ascending order

    avl = AVLTree()
    root = None

    # Insert elements into the AVL tree
    for element in elements:
        root = avl.insert(root, element)

    print("In-order traversal of the AVL tree:")
    avl.in_order(root)
```

4.3 Presentation of the results



4.4 Analysis and discussions

Insertion:

Operation: Inserts a new node into the AVL tree. After the insertion, the tree checks for balance and performs rotations (if necessary) to maintain the AVL property (balance factor between -1 and 1).

Time Complexity: O(logn)

Height Calculation:

Operation: Calculates the height of a node, which is the number of edges on the longest path from that node to a leaf node. The height is updated during insertions and rotations.

Time Complexity: O(1)

In-order Traversal:

Operation: Visits nodes in the order: left subtree, root, right subtree. This results in nodes being printed in ascending order for an AVL tree.

Time Complexity: O(n)

Rotations:

Operation: Rotations are used to maintain the balance of the AVL tree. There are four types of rotations: right rotation, left rotation, left-right rotation, and right-left rotation, depending on the imbalance type.

Time Complexity: O(1)

Experiment 9

Title of the Laboratory Exercise: Quick Sort

1. **Aim**:
   To implement Quick Sort Algorithm using Python

2. **Objective**:
   1. To understand the concept of Quick Sort Algorithm
   2. To learn how to implement Quick Sort Algorithm using Python
   3. To analyze the time complexity of Quick Sort Algorithm

3. **Exercise**:
   In this exercise, you will implement Quick Sort Algorithm using Python. Follow the steps below:
   Step 1: Write a function called quick_sort that takes an array of integers as input and returns a sorted array.
   Step 2: Implement the Quick Sort Algorithm. The steps of the Quick Sort Algorithm are as follows:
   i. Choose a pivot element from the array (can be the first or last element).
   ii. Partition the array into two subarrays: one with elements less than or equal to the pivot, and one with elements greater than the pivot.
   iii. Recursively sort the two subarrays.
   Step 3: Test your implementation using a test case that includes a list of 10 unsorted integers.
   Step 4: Analyze the time complexity of Quick Sort Algorithm.
   Step 5: Submit your code along with a brief explanation of the Quick Sort Algorithm and its time complexity analysis.

   Note: You can use the time module in Python to measure the time taken by your quick_sort function to sort an array.

4. **Experimental Procedure**

   4.1 Algorithm design

```
import time

def quick sort:
  if len <= 1:
    return arr

  pivot = arr[-1]

  if x <= pivot
  left = [x in arr[:-1]]

  if x > pivot
  right = [x in arr[:-1]]

  return quick sort(left) + [pivot] + quick sort(right)

main
```

unsorted array = [12, 7, 5, 9, 3, 11, 1, 4, 10, 8]

start time = time()
sorted array = quick sort(unsorted array)
end time = time()

print(unsorted array)
print(sorted array)
time taken = end_time - start_time:.6f
print(time taken)

4.2 Program

documentation > experiment9-quicksort-ques1.py > quick_sort

```python
import time

# Step 1: Quick Sort function definition
def quick_sort(arr):
    # Base case: if the array has 0 or 1 element, it's already sorted
    if len(arr) <= 1:
        return arr

    # Step 2: Choose a pivot (last element for simplicity)
    pivot = arr[-1]

    # Step 3: Partition the array into two subarrays
    left = [x for x in arr[:-1] if x <= pivot]
    right = [x for x in arr[:-1] if x > pivot]

    # Step 4: Recursively sort the subarrays and combine them with the pivot
    return quick_sort(left) + [pivot] + quick_sort(right)

# Step 3: Test the quick_sort function with an example
if __name__ == "__main__":
    unsorted_array = [12, 7, 5, 9, 3, 11, 1, 4, 10, 8]

    # Measure the time taken to sort the array
    start_time = time.time()
    sorted_array = quick_sort(unsorted_array)
    end_time = time.time()

    print("Unsorted Array: ", unsorted_array)
    print("Sorted Array: ", sorted_array)
    print(f"Time taken to sort the array: {end_time - start_time:.6f} seconds")
```

4.3 Presentation of the results

```
xperiment9-quicksort-ques1.py"
Unsorted Array:  [12, 7, 5, 9, 3, 11, 1, 4, 10, 8]
Sorted Array:  [1, 3, 4, 5, 7, 8, 9, 10, 11, 12]
Time taken to sort the array: 0.000000 seconds
```

4.4 Analysis and discussions

Quick Sort (Main Function):

Operation: Quick sort recursively divides the array into smaller subarrays based on a pivot, sorts the subarrays, and then combines them.

Time Complexity: O(n log n)

Partitioning:

Operation: The array is partitioned into two subarrays: one containing elements less than or equal to the pivot and the other containing elements greater than the pivot.

Time Complexity: O(n)

Recursive Calls:

Operation: After partitioning, quick sort is recursively called on the left and right subarrays.

Time Complexity: O(log n)

List Comprehensions (for partitioning):

Operation: Creates two subarrays (left and right) based on whether the elements are less than or greater than the pivot.

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

Time Measurement:

Operation: Measures the time taken to perform the sorting operation using the time module.

Time Complexity: O(1)