Experiment No: 01

Code:

1. **Write a Program to calculate sum of elements of array [size is not fixed]**

   Program: -

```
ls = []
n = int(input("Enter the number of elements you want:- "))
for i in range(0,n):
    m = int(input(f"Enter the {i+1} element :- "))
    ls.append(m)


sum = 0
for j in ls:
    sum = sum + j
print(f"The Sum of the elements of the array is {sum}")
```

Output :-

```
Enter the number of elements you want:- 5
Enter the 0 element :- 2
Enter the 1 element :- 6
Enter the 2 element :- 8
Enter the 3 element :- 9
Enter the 4 element :- 6
The Sum of the elements of the array is 31
```

---

2. Write a Program to Find Maximum and Minimum value from an array

Program: -

```
ls = []
n = int(input("Enter the number of elements you want:- "))
for i in range(0,n):
    m = int(input(f"Enter the {i+1} element :- "))
    ls.append(m)
max = ls[0]
min = ls[0]
for j in range(0,n):
    if j>max:
        max = ls[j]
    else :
        min = ls[j]
print(f"The maximum element in the array is {max}")
print(f"The minimum element in the array is {min}")
```

Output :

```
Enter the number of elements you want:- 5
Enter the 1 element :- 63
Enter the 2 element :- 69
Enter the 3 element :- 25
Enter the 4 element :- 58
Enter the 5 element :- 14
The maximum element in the array is 63
```

The minimum element in the array is 14

3. Write a Program to sort the elements of array

Program: -

Method Used is Bubble Sort

Method 01

```
ls = []
n = int(input("Enter the number of elements you want:- "))
for i in range(0,n):
    m = int(input(f"Enter the {i+1} element :- "))
    ls.append(m)
print(ls)
ls.sort()
print(ls)
```

Method 02

```
def sorting(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
                swapped = True
        if not swapped:
            break
```

```
ls = []
n = int(input("Enter the number of elements you want:- "))
for i in range(0,n):
    m = int(input(f"Enter the {i+1} element :- "))
    ls.append(m)


sorting(ls)
print("Sorted array is", ls)
```

Output:

Enter the number of elements you want:- 4

Enter the 1 element :- 56

Enter the 2 element :- 45

Enter the 3 element :- 67

Enter the 4 element :- 34

Sorted array is [34, 45, 56, 67]

Experiment No: 02

Code:

```
user_string = input("Enter a Proper String / Sentence:- ")


def reverse_string(s):
    return s[::-1]


def to_uppercase(s):
    return s.upper()


def to_lowercase(s):
    return s.lower()


def count_character(s, char):
    return s.count(char)


def find_substring(s, substring):
    return s.find(substring)  # Returns index of substring or -1 if not found


def is_alpha(s):
    return s.isalpha()


def is_digit(s):
    return s.isdigit()
```

```
print("\nChoose an operation:")
print("1. Reverse the string")
print("2. Convert to uppercase")
print("3. Convert to lowercase")
print("4. Count occurrences of a character")
print("5. Find the index of a substring")
print("6. Check if the string contains only alphabetic characters")
print("7. Check if the string contains only digits")

# Get user choice
choice = int(input("Enter the number of your choice: "))

if choice == 1:
    print("Reversed string:", reverse_string(user_string))

elif choice == 2:
    print("Uppercase string:", to_uppercase(user_string))

elif choice == 3:
    print("Lowercase string:", to_lowercase(user_string))

elif choice == 4:
    char = input("Enter the character to count: ")
    print(f"Occurrences of '{char}':", count_character(user_string, char))

elif choice == 5:
```

```
    substring = input("Enter the substring to find: ")
    index = find_substring(user_string, substring)
    if index != -1:
        print(f"Substring '{substring}' found at index {index}.")
    else:
        print(f"Substring '{substring}' not found.")

elif choice == 6:
    if is_alpha(user_string):
        print("The string contains only alphabetic characters.")
    else:
        print("The string contains non-alphabetic characters.")

elif choice == 7:
    if is_digit(user_string):
        print("The string contains only digits.")
    else:
        print("The string contains non-digit characters.")

else:
    print("Invalid choice. Please select a number between 1 and 7.")
```

Output :-

Enter a Proper String / Sentence:- nutan college of engineering and research

---

Choose an operation:

1. Reverse the string

2. Convert to uppercase

3. Convert to lowercase

4. Count occurrences of a character

5. Find the index of a substring

6. Check if the string contains only alphabetic characters

7. Check if the string contains only digits

Enter the number of your choice: 1

Reversed string: hcraeser dna gnireenigne fo egelloc natun

Enter a Proper String / Sentence:- nutan college of engineering and research

Choose an operation:

1. Reverse the string

2. Convert to uppercase

3. Convert to lowercase

4. Count occurrences of a character

5. Find the index of a substring

6. Check if the string contains only alphabetic characters

7. Check if the string contains only digits

Enter the number of your choice: 2

Uppercase string: NUTAN COLLEGE OF ENGINEERING AND RESEARCH

Enter a Proper String / Sentence:- NUTAN COLLEGE OF ENGINEERING AND RESEARCH

Choose an operation:

---

1. Reverse the string

2. Convert to uppercase

3. Convert to lowercase

4. Count occurrences of a character

5. Find the index of a substring

6. Check if the string contains only alphabetic characters

7. Check if the string contains only digits

Enter the number of your choice: 3

Lowercase string: nutan college of engineering and research

PS C:\WINDOWS\System32\WindowsPowerShell\v1.0> python -u "d:\[BASICS 64GB]\[Current Learning]\TY NOTES\CP\Practical 03\Practical_03.py"

Enter a Proper String / Sentence:- There is a black dog beside a brown dog who are teamed with white dog

Choose an operation:

1. Reverse the string

2. Convert to uppercase

3. Convert to lowercase

4. Count occurrences of a character

5. Find the index of a substring

6. Check if the string contains only alphabetic characters

7. Check if the string contains only digits

Enter the number of your choice: 4

Enter the character to count: dog

Occurrences of 'dog': 3

PS C:\WINDOWS\System32\WindowsPowerShell\v1.0>

Experiment No: 03

Code:

Program 01: Linear Search

```
#Linear_Search


def linear_search(arr,target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1


arr = [1,2,3,4,5,6]
target = 5
result = linear_search(arr,target)


if result != -1:
    print("Element is present at index",(result))
else:
    print("Linear Search Element Not Found")
```

Output:

Element is present at index 4

Program 02: Bubble Sort

```
#Bubble_Sort
def bubble_sort(arr):
    n = len(arr)
```

```
    for i in range(n - 1, 0, -1):
        swapped = False
        for j in range(i):
            if arr[j] > arr[j + 1]:
                swapped = True
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
        if not swapped:
            break  # Use 'break' to exit the loop if no elements were swapped


arr = [34,67,89,43,2,11,99,78]
print("Unsorted list is:")
print(arr)
bubble_sort(arr)
print("Sorted list is:")
print(arr)
```

Output:

Unsorted list is:

[34, 67, 89, 43, 2, 11, 99, 78]

Sorted list is:

[2, 11, 34, 43, 67, 78, 89, 99]

Program 03: Insertion Sort

```
#Insertion_Sort
def insertion_sort(arr):
    n = len(arr)
```

```
    if n<=1:
        return

    for i in range(1,n):
        key = arr[i]
        j=i-1
        while j>=0 and key<arr[j] :
            arr[j+1]==arr[j]
            j-=1
        arr[j+1]=key

arr = [45,67,89,34,56,2,4,6]
print("Unsorted list is:")
print(arr)

insertion_sort(arr)

print("Sorted list is:")
print(arr)
```

Output:

Unsorted list is:

[45, 67, 89, 34, 56, 2, 4, 6]

Sorted list is:

[2, 67, 89, 34, 56, 2, 4, 6]

Program 04: Binary Search

```
#Binary_Search
def binary_search(arr, target, low, high):
    if low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            return binary_search(arr, target, mid + 1, high)
        else:
            return binary_search(arr, target, low, mid - 1)
    else:
        return -1


arr = [4,6,8,5,3,9,10]
target = 10
result = binary_search(arr, target, 0, len(arr) - 1)
if result !=-1:
    print(f"Binary Search Element found at {result}")
else:
    print("Element not found in array")
```

Output:

Binary Search Element found at 6

Experiment No: 04

Code:

Program 01: Addition of Matrices

```
# Addition of Matrices

# Input the number of matrices
num_matrices = int(input("Enter the number of matrices you want to add: "))

# Input the number of rows and columns for the matrices
X = int(input("Enter the number of rows: "))
Y = int(input("Enter the number of columns: "))

# Function to take matrix input from the user
def input_matrix(matrix_num):
    matrix = []
    print(f"Enter the entries for matrix {matrix_num} row by row:")
    for i in range(X):
        row = []
        for j in range(Y):
            row.append(int(input(f"Enter element [{i+1},{j+1}]: ")))
        matrix.append(row)
    return matrix

# Store all matrices in a list
matrices = []
```

```
# Loop to input all matrices
for m in range(1, num_matrices + 1):
    matrix = input_matrix(m)
    matrices.append(matrix)

# Display all matrices
for m in range(num_matrices):
    print(f"\nMatrix {m + 1}:")
    for row in matrices[m]:
        print(row)

# Initialize the result matrix with zeros
result = [[0 for _ in range(Y)] for _ in range(X)]

# Add all matrices element by element
for matrix in matrices:
    for i in range(X):
        for j in range(Y):
            result[i][j] += matrix[i][j]

# Display the result matrix
print("\nResultant Matrix after addition:")
for row in result:
    print(row)
```

Output:

Enter the number of matrices you want to add: 2

Enter the number of rows: 3

Enter the number of columns: 3

Enter the entries for matrix 1 row by row:

Enter element [1,1]: 3

Enter element [1,2]: 4

Enter element [1,3]: 5

Enter element [2,1]: 6

Enter element [2,2]: 7

Enter element [2,3]: 8

Enter element [3,1]: 1

Enter element [3,2]: 2

Enter element [3,3]: 3

Enter the entries for matrix 2 row by row:

Enter element [1,1]: 5

Enter element [1,2]: 6

Enter element [1,3]: 7

Enter element [2,1]: 8

Enter element [2,2]: 9

Enter element [2,3]: 1

Enter element [3,1]: 2

Enter element [3,2]: 3

Enter element [3,3]: 4

Matrix 1:

[3, 4, 5]

[6, 7, 8]

[1, 2, 3]

Matrix 2:

[5, 6, 7]

[8, 9, 1]

[2, 3, 4]

Resultant Matrix after addition:

[8, 10, 12]

[14, 16, 9]

[3, 5, 7]

Program 02: Multiplication of Matrices

```
#Multiplcation of the Matrices

# Input the number of matrices
num_matrices = int(input("Enter the number of matrices you want to multiply: "))

# Input the dimensions for the first matrix
A_rows = int(input("Enter the number of rows for the first matrix: "))
A_cols = int(input("Enter the number of columns for the first matrix: "))

# Function to input matrix elements from the user
def input_matrix(matrix_num, rows, cols):
    matrix = []
```

```
    print(f"Enter the entries for matrix {matrix_num} row by row:")
    for i in range(rows):
        row = []
        for j in range(cols):
            row.append(int(input(f"Enter element [{i+1},{j+1}]: ")))
        matrix.append(row)
    return matrix

# Store all matrices
matrices = []

# Input the first matrix
matrix = input_matrix(1, A_rows, A_cols)
matrices.append(matrix)

# Now we need to input remaining matrices and check compatibility for multiplication
for m in range(2, num_matrices + 1):
    # For the m-th matrix, input rows and columns
    B_rows = int(input(f"Enter the number of rows for matrix {m}: "))
    B_cols = int(input(f"Enter the number of columns for matrix {m}: "))

    # Check if the current matrix is compatible with the previous matrix for multiplication
    if A_cols != B_rows:
        print(f"Matrix multiplication cannot be performed because the number of columns of matrix {m-1} "
              "is not equal to the number of rows of matrix {m}.")
```

```
        exit()

    # Input the m-th matrix
    matrix = input_matrix(m, B_rows, B_cols)
    matrices.append(matrix)

    # Update dimensions for the next matrix multiplication
    A_rows = B_rows
    A_cols = B_cols

# Display the matrices entered by the user
for m in range(num_matrices):
    print(f"\nMatrix {m + 1}:")
    for row in matrices[m]:
        print(row)

# Function to multiply two matrices
def multiply_matrices(A, B):
    # Get the dimensions of A and B
    A_rows = len(A)
    A_cols = len(A[0])
    B_rows = len(B)
    B_cols = len(B[0])

    # Check if multiplication is possible
    if A_cols != B_rows:
```

```
        print("Matrix multiplication is not possible due to incompatible dimensions.")
        return None

    # Initialize the result matrix with zeros
    result = [[0 for _ in range(B_cols)] for _ in range(A_rows)]

    # Perform matrix multiplication
    for i in range(A_rows):
        for j in range(B_cols):
            for k in range(A_cols):
                result[i][j] += A[i][k] * B[k][j]

    return result

# Multiply all matrices
result_matrix = matrices[0]
for i in range(1, num_matrices):
    result_matrix = multiply_matrices(result_matrix, matrices[i])

# Display the result of multiplication
print("\nResultant Matrix after multiplication:")
for row in result_matrix:
    print(row)
```

Output:

Enter the number of matrices you want to multiply: 2

Enter the number of rows for the first matrix: 2

Enter the number of columns for the first matrix: 2

Enter the entries for matrix 1 row by row:

Enter element [1,1]: 2

Enter element [1,2]: 3

Enter element [2,1]: 4

Enter element [2,2]: 5

Enter the number of rows for matrix 2: 2

Enter the number of columns for matrix 2: 2

Enter the entries for matrix 2 row by row:

Enter element [1,1]: 8

Enter element [1,2]: 7

Enter element [2,1]: 6

Enter element [2,2]: 5

Matrix 1:

[2, 3]

[4, 5]

Matrix 2:

[8, 7]

[6, 5]

Resultant Matrix after multiplication:

[34, 29]

[62, 53]

Program 03: Transpose of the Matrix

```
# Input the number of rows and columns for the matrix
X = int(input("Enter the number of rows: "))
Y = int(input("Enter the number of columns: "))

# Function to take matrix input from the user
def input_matrix():
    matrix = []
    print("Enter the entries for the matrix row by row:")
    for i in range(X):
        row = []
        for j in range(Y):
            row.append(int(input(f"Enter element [{i+1},{j+1}]: ")))
        matrix.append(row)
    return matrix

# Input the matrix
matrix = input_matrix()

# Display the original matrix
print("\nOriginal Matrix:")
for row in matrix:
    print(row)

# Compute the transpose of the matrix
transpose = [[matrix[i][j] for i in range(X)] for j in range(Y)]
```

```
# Display the transposed matrix
print("\nTransposed Matrix:")
for row in transpose:
    print(row)
```

Output:

```
Enter the number of rows: 3
Enter the number of columns: 3
Enter the entries for the matrix row by row:
Enter element [1,1]: 3
Enter element [1,2]: 4
Enter element [1,3]: 5
Enter element [2,1]: 6
Enter element [2,2]: 7
Enter element [2,3]: 8
Enter element [3,1]: 2
Enter element [3,2]: 3
Enter element [3,3]: 4
Original Matrix:
[3, 4, 5]
[6, 7, 8]
[2, 3, 4]
Transposed Matrix:
[3, 6, 2]
[4, 7, 3]
[5, 8, 4]
```

Experiment No: 05

Code:

Program: Implementation of Linked List along with its operations

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


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


    # Method to add a node at begin of LL
    def insertAtBegin(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
        else:
            new_node.next = self.head
            self.head = new_node


    # Method to add a node at any index (Indexing starts from 0)
    def insertAtIndex(self, data, index):
        if index == 0:
            self.insertAtBegin(data)
```

```
            return
        position = 0
        current_node = self.head
        while current_node is not None and position + 1 != index:
            position += 1
            current_node = current_node.next
        if current_node is not None:
            new_node = Node(data)
            new_node.next = current_node.next
            current_node.next = new_node
        else:
            print("Index not present")

    # Method to add a node at the end of LL
    def insertAtEnd(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            return
        current_node = self.head
        while current_node.next:
            current_node = current_node.next
        current_node.next = new_node

    # Method to remove first node of linked list
    def remove_first_node(self):
```

```
        if self.head is None:
            return
        self.head = self.head.next

    # Method to remove last node of linked list
    def remove_last_node(self):
        if self.head is None:
            return
        current_node = self.head
        while current_node.next and current_node.next.next:
            current_node = current_node.next
        current_node.next = None

    # Method to remove at given index
    def remove_at_index(self, index):
        if self.head is None:
            return
        current_node = self.head
        position = 0
        if position == index:
            self.remove_first_node()
            return
        while current_node and position + 1 != index:
            position += 1
            current_node = current_node.next
        if current_node and current_node.next:
```

```
            current_node.next = current_node.next.next
        else:
            print("Index not present")

    # Method to remove a node from linked list by data
    def remove_node(self, data):
        current_node = self.head
        if current_node.data == data:
            self.remove_first_node()
            return
        while current_node and current_node.next and current_node.next.data != data:
            current_node = current_node.next
        if current_node and current_node.next:
            current_node.next = current_node.next.next

    # Print method for the linked list
    def printLL(self):
        if self.head is None:
            print("The list is empty.")
            return
        current_node = self.head
        while current_node:
            print(current_node.data, end=" -> ")
            current_node = current_node.next
        print("None")
```

```
# Create a new linked list
llist = LinkedList()

# Function to handle user input and operations
def menu():
    while True:
        # Display the menu options
        print("\nLinked List Operations:")
        print("1. Insert at Beginning")
        print("2. Insert at Index")
        print("3. Insert at End")
        print("4. Remove First Node")
        print("5. Remove Last Node")
        print("6. Remove Node at Index")
        print("7. Remove Node by Data")
        print("8. Print Linked List")
        print("9. Exit")

        # Display the current state of the linked list before taking action
        print("\nCurrent Linked List:")
        llist.printLL()

        # Get user choice
        choice = input("Enter your choice: ").strip()

        if choice == '1':
```

```
        data = input("Enter the data to insert at beginning: ").strip()
        llist.insertAtBegin(data)
    elif choice == '2':
        data = input("Enter the data to insert at index: ").strip()
        index = int(input("Enter the index: ").strip())
        llist.insertAtIndex(data, index)
    elif choice == '3':
        data = input("Enter the data to insert at end: ").strip()
        llist.insertAtEnd(data)
    elif choice == '4':
        llist.remove_first_node()
    elif choice == '5':
        llist.remove_last_node()
    elif choice == '6':
        index = int(input("Enter the index to remove: ").strip())
        llist.remove_at_index(index)
    elif choice == '7':
        data = input("Enter the data to remove: ").strip()
        llist.remove_node(data)
    elif choice == '8':
        print("Current Linked List:")
        llist.printLL()
    elif choice == '9':
        print("Exiting the program.")
        break
    else:
```

```
        print("Invalid choice, please try again.")


# Run the menu function to interact with the user

menu()
```

Output:

```
Linked List Operations:

1. Insert at Beginning

2. Insert at Index

3. Insert at End

4. Remove First Node

5. Remove Last Node

6. Remove Node at Index

7. Remove Node by Data

8. Print Linked List

9. Exit


Current Linked List:

The list is empty.

Enter your choice: 1

Enter the data to insert at beginning: 23


Linked List Operations:

1. Insert at Beginning

2. Insert at Index

3. Insert at End

4. Remove First Node
```

5. Remove Last Node

6. Remove Node at Index

7. Remove Node by Data

8. Print Linked List

9. Exit

Current Linked List:

23 -> None

Enter your choice: 2

Enter the data to insert at index: 34

Enter the index: 1

Linked List Operations:

1. Insert at Beginning

2. Insert at Index

3. Insert at End

4. Remove First Node

5. Remove Last Node

6. Remove Node at Index

7. Remove Node by Data

8. Print Linked List

9. Exit

Current Linked List:

23 -> 34 -> None

Enter your choice: 3

Enter the data to insert at end: 45

Linked List Operations:

1. Insert at Beginning
2. Insert at Index
3. Insert at End
4. Remove First Node
5. Remove Last Node
6. Remove Node at Index
7. Remove Node by Data
8. Print Linked List
9. Exit

Current Linked List:

23 -> 34 -> 45 -> None

Enter your choice: 4

Linked List Operations:

1. Insert at Beginning
2. Insert at Index
3. Insert at End
4. Remove First Node
5. Remove Last Node
6. Remove Node at Index
7. Remove Node by Data
8. Print Linked List

9. Exit

Current Linked List:

34 -> 45 -> None

Enter your choice: 5

Linked List Operations:

1. Insert at Beginning
2. Insert at Index
3. Insert at End
4. Remove First Node
5. Remove Last Node
6. Remove Node at Index
7. Remove Node by Data
8. Print Linked List
9. Exit

Current Linked List:

34 -> None

Enter your choice: 9

Exiting the program.

Experiment No: 06

Code:

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

# Function to build a binary tree from user input
def build_tree():
    num_nodes = int(input("Enter the number of nodes: "))
    if num_nodes == 0:
        return None
    root_data = input("Enter the root node value: ")
    if root_data.lower() == 'none':
        return None
    root = Node(int(root_data))
    queue = [root]
    count = 1

    while queue and count < num_nodes:
        current = queue.pop(0)
        left_data = input(f"Enter left child of {current.data} (or 'None'): ")
        if left_data.lower() != 'none':
            current.left = Node(int(left_data))
```

```
            queue.append(current.left)
            count += 1
        if count >= num_nodes:
            break
        right_data = input(f"Enter right child of {current.data} (or 'None'): ")
        if right_data.lower() != 'none':
            current.right = Node(int(right_data))
            queue.append(current.right)
            count += 1
    return root

# Function to check if the binary tree is complete
def is_complete_btree(root):
    if not root:
        return True
    queue = [root]
    found_null = False

    while queue:
        current = queue.pop(0)
        if current:
            if found_null:
                return False
            queue.append(current.left)
            queue.append(current.right)
        else:
```

```
        found_null = True
    return True


# Function to check if two nodes are cousins
def are_cousins(root, node1, node2):
    def find_parent_and_level(root, node, level=0):
        if not root:
            return None, 0
        if (root.left and root.left.data == node) or (root.right and root.right.data == node):
            return root, level + 1
        left_parent, left_level = find_parent_and_level(root.left, node, level + 1)
        if left_parent:
            return left_parent, left_level
        return find_parent_and_level(root.right, node, level + 1)


    parent1, level1 = find_parent_and_level(root, node1)
    parent2, level2 = find_parent_and_level(root, node2)


    return level1 == level2 and parent1 != parent2


# Driver Code
root = build_tree()


# Check if the tree is complete
if is_complete_btree(root):
    print("Complete Binary Tree")
```

```
else:
    print("NOT Complete Binary Tree")


# Check if two nodes are cousins
node1 = int(input("Enter the first node to check if it is a cousin: "))
node2 = int(input("Enter the second node to check if it is a cousin: "))
if are_cousins(root, node1, node2):
    print(f"Nodes {node1} and {node2} are cousins.")
else:
    print(f"Nodes {node1} and {node2} are NOT cousins.")
```

Output:

```
Enter the number of nodes: 6
Enter the root node value: 4
Enter left child of 4 (or 'None'): 5
Enter right child of 4 (or 'None'): 6
Enter left child of 5 (or 'None'): 7
Enter right child of 5 (or 'None'): 8
Enter left child of 6 (or 'None'): 9
Complete Binary Tree
Enter the first node to check if it is a cousin: 4
Enter the second node to check if it is a cousin: 6
Nodes 4 and 6 are NOT cousins.
```

Experiment No: 07

Code:

Program 01: To Check whether the given tree is binary tree or not

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


# Function to build a binary tree from user input
def build_tree():
    num_nodes = int(input("Enter the number of nodes: "))
    if num_nodes == 0:
        return None
    root_data = input("Enter the root node value: ")
    if root_data.lower() == 'none':
        return None
    root = Node(int(root_data))
    queue = [root]
    count = 1

    while queue and count < num_nodes:
        current = queue.pop(0)
        left_data = input(f"Enter left child of {current.data} (or 'None'): ")
        if left_data.lower() != 'none':
```

```
            current.left = Node(int(left_data))
            queue.append(current.left)
            count += 1
        if count >= num_nodes:
            break
        right_data = input(f"Enter right child of {current.data} (or 'None'): ")
        if right_data.lower() != 'none':
            current.right = Node(int(right_data))
            queue.append(current.right)
            count += 1
    return root

# Function to check if the tree is a valid binary tree
def is_binary_tree(root):
    if not root:
        return True  # An empty tree is considered a valid binary tree

    # Helper function to check if each node has at most two children
    def check_children(node):
        if not node:
            return True
        # A node should not have more than two children (left and right)
        # Just ensuring that each node has at most one left and one right child
        return (check_children(node.left) and check_children(node.right))

    # Call the helper function on the root node
```

```
    return check_children(root)


# Driver Code
root = build_tree()


# Check if the tree is a valid binary tree
if is_binary_tree(root):
    print("The tree is a valid Binary Tree.")
else:
    print("The tree is NOT a valid Binary Tree.")
```

Output:

```
Enter the number of nodes: 6
Enter the root node value: 1
Enter left child of 1 (or 'None'): 2
Enter right child of 1 (or 'None'): 3
Enter left child of 2 (or 'None'): 4
Enter right child of 2 (or 'None'): 4
Enter left child of 3 (or 'None'): 5
The tree is a valid Binary Tree.
```

Program 02: Binary Search Tree

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

```
# Function to insert a node in the Binary Search Tree
def insert(root, value):
    # If the tree is empty, create a new node
    if root is None:
        return Node(value)

    # Otherwise, recur down the tree
    if value < root.data:
        root.left = insert(root.left, value)  # Insert in the left subtree
    elif value > root.data:
        root.right = insert(root.right, value)  # Insert in the right subtree

    return root

# Function to search a value in the Binary Search Tree
def search(root, value):
    # Base Case: root is null or value is present at the root
    if root is None or root.data == value:
        return root

    # Value is greater than root's data, search in the right subtree
    if value > root.data:
        return search(root.right, value)

    # Value is smaller than root's data, search in the left subtree
```

```
    return search(root.left, value)


# Function to build the Binary Search Tree from user input
def build_bst():
    num_nodes = int(input("Enter the number of nodes in the BST: "))
    if num_nodes == 0:
        return None

    # Get the root node
    root_value = int(input("Enter the root node value: "))
    root = Node(root_value)

    # Insert other nodes into the BST
    for _ in range(num_nodes - 1):
        value = int(input("Enter a value to insert in the BST: "))
        root = insert(root, value)

    return root

# Main Driver Code
root = build_bst()

# Asking the user to enter a value to search in the Binary Search Tree
search_value = int(input("Enter a value to search in the BST: "))

# Searching the value in the BST
```

```
if search(root, search_value):
    print(f"Value {search_value} is present in the Binary Search Tree.")
else:
    print(f"Value {search_value} is NOT present in the Binary Search Tree.")
```

Output:

Enter the number of nodes in the BST: 6

Enter the root node value: 4

Enter a value to insert in the BST: 5

Enter a value to insert in the BST: 6

Enter a value to insert in the BST: 7

Enter a value to insert in the BST: 8

Enter a value to insert in the BST: 9

Enter a value to search in the BST: 11

Value 11 is NOT present in the Binary Search Tree.

Enter the number of nodes in the BST: 6

Enter the root node value: 1

Enter a value to insert in the BST: 2

Enter a value to insert in the BST: 3

Enter a value to insert in the BST: 4

Enter a value to insert in the BST: 5

Enter a value to insert in the BST: 6

Enter a value to search in the BST: 5

Value 5 is present in the Binary Search Tree.

Experiment No: 08

Code:

Program 01: Implement the stack by using a linked list and display its values. Perform all the operations related to the stack.

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


class Stack:
    def __init__(self):
        self.top = None

    # Check if the stack is empty
    def is_empty(self):
        return self.top is None

    # Push a value onto the stack
    def push(self, data):
        new_node = Node(data)
        new_node.next = self.top
        self.top = new_node
        print(f"Pushed {data} onto the stack.")

    # Pop a value from the stack
    def pop(self):
```

```
        if self.is_empty():
            print("Stack Underflow! Cannot pop from an empty stack.")
            return None
        popped_node = self.top
        self.top = self.top.next
        print(f"Popped {popped_node.data} from the stack.")
        return popped_node.data

    # Peek at the top value of the stack
    def peek(self):
        if self.is_empty():
            print("Stack is empty.")
            return None
        print(f"Top element is {self.top.data}.")
        return self.top.data

    # Display all elements in the stack
    def display(self):
        if self.is_empty():
            print("Stack is empty.")
            return
        current = self.top
        print("Stack elements are:")
        while current:
            print(current.data, end=" -> ")
            current = current.next
```

```
        print("None")


# Main program with user menu
def main():
    stack = Stack()

    while True:
        print("\nSelect an operation:")
        print("1. Push")
        print("2. Pop")
        print("3. Peek")
        print("4. Display")
        print("5. Check if Empty")
        print("6. Exit")

        choice = input("Enter your choice (1-6): ")

        if choice == '1':
            data = int(input("Enter the value to push: "))
            stack.push(data)
        elif choice == '2':
            stack.pop()
        elif choice == '3':
            stack.peek()
        elif choice == '4':
            stack.display()
```

```
        elif choice == '5':
            if stack.is_empty():
                print("Stack is empty.")
            else:
                print("Stack is not empty.")
        elif choice == '6':
            print("Exiting program.")
            break
        else:
            print("Invalid choice. Please select a valid option.")


if __name__ == "__main__":
    main()
```

Output:

Select an operation:

1. Push

2. Pop

3. Peek

4. Display

5. Check if Empty

6. Exit

Enter your choice (1-6): 1

Enter the value to push: 23

Pushed 23 onto the stack.

Select an operation:

1. Push

2. Pop

3. Peek

4. Display

5. Check if Empty

6. Exit

Enter your choice (1-6): 1

Enter the value to push: 45

Pushed 45 onto the stack.

Select an operation:

1. Push

2. Pop

3. Peek

4. Display

5. Check if Empty

6. Exit

Enter your choice (1-6): 2

Popped 45 from the stack.

Select an operation:

1. Push

2. Pop

3. Peek

4. Display

5. Check if Empty

---

6. Exit

Enter your choice (1-6): 3

Top element is 23.

Select an operation:

1. Push

2. Pop

3. Peek

4. Display

5. Check if Empty

6. Exit

Enter your choice (1-6): 4

Stack elements are:

23 -> None

Select an operation:

1. Push

2. Pop

3. Peek

4. Display

5. Check if Empty

6. Exit

Enter your choice (1-6): 5

Stack is not empty.

Select an operation:

```
1. Push
2. Pop
3. Peek
4. Display
5. Check if Empty
6. Exit
Enter your choice (1-6): 6
Exiting program.
```

Program 02: Implement a Circular Queue by using an array. Perform all the operations related to the circular queue.

```
class CircularQueue:
    def __init__(self, size):
        self.size = size
        self.queue = [None] * size
        self.front = -1
        self.rear = -1

    # Check if the queue is empty
    def is_empty(self):
        return self.front == -1

    # Check if the queue is full
    def is_full(self):
        return (self.rear + 1) % self.size == self.front

    # Enqueue an element into the queue
```

```
    def enqueue(self, data):
        if self.is_full():
            print("Queue Overflow! Cannot enqueue because the queue is full.")
            return
        if self.is_empty():
            self.front = 0
        self.rear = (self.rear + 1) % self.size
        self.queue[self.rear] = data
        print(f"Enqueued {data} to the queue.")

    # Dequeue an element from the queue
    def dequeue(self):
        if self.is_empty():
            print("Queue Underflow! Cannot dequeue because the queue is empty.")
            return None
        data = self.queue[self.front]
        if self.front == self.rear:  # Only one element in the queue
            self.front = self.rear = -1
        else:
            self.front = (self.front + 1) % self.size
        print(f"Dequeued {data} from the queue.")
        return data

    # Peek at the front element of the queue
    def peek(self):
        if self.is_empty():
```

```
            print("Queue is empty. Nothing to peek.")
            return None
        print(f"Front element is {self.queue[self.front]}.")
        return self.queue[self.front]

    # Display all elements in the queue
    def display(self):
        if self.is_empty():
            print("Queue is empty.")
            return
        print("Queue elements are:", end=" ")
        i = self.front
        while True:
            print(self.queue[i], end=" ")
            if i == self.rear:
                break
            i = (i + 1) % self.size
        print()

# Example usage of the CircularQueue class
if __name__ == "__main__":
    size = int(input("Enter the size of the circular queue: "))
    queue = CircularQueue(size)

    # Perform queue operations
    while True:
```

```
print("\nSelect an operation:")
print("1. Enqueue")
print("2. Dequeue")
print("3. Peek")
print("4. Display")
print("5. Check if Empty")
print("6. Check if Full")
print("7. Exit")

choice = input("Enter your choice (1-7): ")

if choice == '1':
    data = int(input("Enter the value to enqueue: "))
    queue.enqueue(data)
elif choice == '2':
    queue.dequeue()
elif choice == '3':
    queue.peek()
elif choice == '4':
    queue.display()
elif choice == '5':
    if queue.is_empty():
        print("Queue is empty.")
    else:
        print("Queue is not empty.")
elif choice == '6':
```

```
        if queue.is_full():
            print("Queue is full.")
        else:
            print("Queue is not full.")
    elif choice == '7':
        print("Exiting program.")
        break
    else:
        print("Invalid choice. Please select a valid option.")
```

Output:

Enter the size of the circular queue: 5

Select an operation:

1. Enqueue

2. Dequeue

3. Peek

4. Display

5. Check if Empty

6. Check if Full

7. Exit

Enter your choice (1-7): 1

Enter the value to enqueue: 34

Enqueued 34 to the queue.

Select an operation:

1. Enqueue

2. Dequeue

3. Peek

4. Display

5. Check if Empty

6. Check if Full

7. Exit

Enter your choice (1-7): 1

Enter the value to enqueue: 34

Enqueued 34 to the queue.

Select an operation:

1. Enqueue

2. Dequeue

3. Peek

4. Display

5. Check if Empty

6. Check if Full

7. Exit

Enter your choice (1-7): 1

Enter the value to enqueue: 56

Enqueued 56 to the queue.

Select an operation:

1. Enqueue

2. Dequeue

3. Peek

4. Display

5. Check if Empty

6. Check if Full

7. Exit

Enter your choice (1-7): 2

Dequeued 34 from the queue.

Select an operation:

1. Enqueue

2. Dequeue

3. Peek

4. Display

5. Check if Empty

6. Check if Full

7. Exit

Enter your choice (1-7): 3

Front element is 34.

Select an operation:

1. Enqueue

2. Dequeue

3. Peek

4. Display

5. Check if Empty

6. Check if Full

7. Exit

Enter your choice (1-7): 4

Queue elements are: 34 56

Select an operation:

1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Check if Empty
6. Check if Full
7. Exit

Enter your choice (1-7): 5

Queue is not empty.

Select an operation:

1. Enqueue
2. Dequeue
3. Peek
4. Display
5. Check if Empty
6. Check if Full
7. Exit

Enter your choice (1-7): 6

Queue is not full.

Select an operation:

1. Enqueue

2. Dequeue

3. Peek

4. Display

5. Check if Empty

6. Check if Full

7. Exit

Enter your choice (1-7): 7

Exiting program.

Experiment No: 09

Code:

Program 01: To Check whether a binary tree is min heap or not

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

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

    # Insert nodes level-wise to maintain a complete binary tree
    def insert(self, data):
        new_node = Node(data)
        if self.root is None:
            self.root = new_node
            return
        queue = [self.root]
        while queue:
            node = queue.pop(0)
            if not node.left:
                node.left = new_node
                return
```

```
            else:
                queue.append(node.left)
            if not node.right:
                node.right = new_node
                return
            else:
                queue.append(node.right)

    # Count the total number of nodes in the tree
    def count_nodes(self, node):
        if node is None:
            return 0
        return 1 + self.count_nodes(node.left) + self.count_nodes(node.right)

    # Check if the tree is complete
    def is_complete(self, node, index, node_count):
        if node is None:
            return True
        if index >= node_count:
            return False
        return (self.is_complete(node.left, 2 * index + 1, node_count) and
                self.is_complete(node.right, 2 * index + 2, node_count))

    # Check if the tree follows the min-heap property
    def is_min_heap_property(self, node):
        if node is None:
```

```
            return True

        # If node has no children
        if not node.left and not node.right:
            return True

        # If node has only left child
        if node.left and not node.right:
            return node.data <= node.left.data and self.is_min_heap_property(node.left)

        # If node has both children
        if node.left and node.right:
            return (node.data <= node.left.data and
                    node.data <= node.right.data and
                    self.is_min_heap_property(node.left) and
                    self.is_min_heap_property(node.right))

    # Check if the binary tree is a min-heap
    def is_min_heap(self):
        node_count = self.count_nodes(self.root)
        if self.is_complete(self.root, 0, node_count) and self.is_min_heap_property(self.root):
            return True
        return False

# Drive Code
if __name__ == "__main__":
```

```
    bt = BinaryTree()

    n = int(input("Enter the number of elements in the binary tree: "))

    print("Enter the elements:")

    for _ in range(n):

        element = int(input())

        bt.insert(element)


    if bt.is_min_heap():

        print("The binary tree is a min-heap.")

    else:

        print("The binary tree is not a min-heap.")
```

Output:

Enter the number of elements in the binary tree: 6

Enter the elements:

1

2

3

4

5

6

The binary tree is a min-heap.

Program 02: To Check whether a binary tree is max heap or not

```
class Node:

    def __init__(self, data):

        self.data = data
```

```
        self.left = None
        self.right = None


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


    # Insert nodes level-wise to maintain a complete binary tree
    def insert(self, data):
        new_node = Node(data)
        if self.root is None:
            self.root = new_node
            return
        queue = [self.root]
        while queue:
            node = queue.pop(0)
            if not node.left:
                node.left = new_node
                return
            else:
                queue.append(node.left)
            if not node.right:
                node.right = new_node
                return
            else:
                queue.append(node.right)
```

```
    # Count the total number of nodes in the tree
    def count_nodes(self, node):
        if node is None:
            return 0
        return 1 + self.count_nodes(node.left) + self.count_nodes(node.right)

    # Check if the tree is complete
    def is_complete(self, node, index, node_count):
        if node is None:
            return True
        if index >= node_count:
            return False
        return (self.is_complete(node.left, 2 * index + 1, node_count) and
                self.is_complete(node.right, 2 * index + 2, node_count))

    # Check if the tree follows the max-heap property
    def is_max_heap_property(self, node):
        if node is None:
            return True

        # If node has no children
        if not node.left and not node.right:
            return True

        # If node has only left child
```

```
        if node.left and not node.right:
            return node.data >= node.left.data and self.is_max_heap_property(node.left)

        # If node has both children
        if node.left and node.right:
            return (node.data >= node.left.data and
                    node.data >= node.right.data and
                    self.is_max_heap_property(node.left) and
                    self.is_max_heap_property(node.right))

    # Check if the binary tree is a max-heap
    def is_max_heap(self):
        node_count = self.count_nodes(self.root)
        if self.is_complete(self.root, 0, node_count) and self.is_max_heap_property(self.root):
            return True
        return False

# Drive Code
if __name__ == "__main__":
    bt = BinaryTree()
    n = int(input("Enter the number of elements in the binary tree: "))
    print("Enter the elements:")
    for _ in range(n):
        element = int(input())
        bt.insert(element)
```

```
    if bt.is_max_heap():
        print("The binary tree is a max-heap.")
    else:
        print("The binary tree is not a max-heap.")
```

Output:

Enter the number of elements in the binary tree: 6

Enter the elements:

10

9

8

7

6

5

The binary tree is a max-heap.

Experiment No: 10

Code:

```
# Fractional Knapsack Problem using the Greedy Algorithm.
class Item:
    def __init__(self, value, weight):
        self.value = value
        self.weight = weight
# Function to calculate the maximum value of items in the knapsack
def fractional_knapsack(capacity, items):
    # Calculate value-to-weight ratio for each item
    items = sorted(items, key=lambda x: x.value / x.weight, reverse=True)

    total_value = 0.0  # Total value accumulated in the knapsack
    for item in items:
        if capacity == 0:  # If the knapsack is full, break
            break
        # If item can be fully added to the knapsack
        if item.weight <= capacity:
            capacity -= item.weight
            total_value += item.value
            print(f"Taking full item with value {item.value} and weight {item.weight}")
        else:
            # Take fraction of the remaining capacity
            fraction = capacity / item.weight
            total_value += item.value * fraction
```

```
            print(f"Taking {fraction * 100:.2f}% of item with value {item.value} and weight {item.weight}")
            capacity = 0  # Knapsack is now full
    return total_value


# Driver Code
if __name__ == "__main__":
    n = int(input("Enter the number of items: "))
    items = []
    for i in range(n):
        value = float(input(f"Enter value of item {i+1}: "))
        weight = float(input(f"Enter weight of item {i+1}: "))
        items.append(Item(value, weight))
    capacity = float(input("Enter the maximum weight capacity of the knapsack: "))
    max_value = fractional_knapsack(capacity, items)
    print(f"\nMaximum value in the knapsack: {max_value}")
```

Output:

Enter the number of items: 4

Enter value of item 1: 5

Enter weight of item 1: 6

Enter value of item 2: 1

Enter weight of item 2: 4

Enter value of item 3: 3

Enter weight of item 3: 6

Enter value of item 4: 4

Enter weight of item 4: 7

Enter the maximum weight capacity of the knapsack: 20

Taking full item with value 5.0 and weight 6.0

Taking full item with value 4.0 and weight 7.0

Taking full item with value 3.0 and weight 6.0

Taking 25.00% of item with value 1.0 and weight 4.0

Maximum value in the knapsack: 12.25