

## Week 1

### Topic : To Implement Sorting Algorithm

1. Write a Python program to implement the concept of Bubble sort on the above data set. Print the data set after every iteration.

#### Code:

```
def bubble_sort(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

        print(f"After iteration {i+1}: {arr}")

        if not swapped:
            break

data = [27,15,39,21,28,70]

bubble_sort(data)
```

#### Output:

```
PS C:\Users\User\Desktop\python_ > & C:/Users/User/anaconda3/python.exe c:/Users/User/Desktop/python/sample.py
After iteration 1: [15, 27, 21, 28, 39, 70]
After iteration 2: [15, 21, 27, 28, 39, 70]
After iteration 3: [15, 21, 27, 28, 39, 70]
PS C:\Users\User\Desktop\python_ > []
```

2. Write a Python program to implement the concept of selection sort on the above data set. Print the data set after every iteration.

**Code:**

```
def selection_sort(arr):
    n = len(arr)

    for i in range(n):
        min_index = i
        for j in range(i+1, n):
            if arr[j] < arr[min_index]:
                min_index = j

        arr[i], arr[min_index] = arr[min_index], arr[i]

        print(f"After iteration {i+1}: {arr}")

data = [27,15,39,21,28,70]
selection_sort(data)
```

**Output:**

```
PS C:\Users\User\Desktop\python_ > & C:/Users/User/anaconda3/python.exe c:/Users/User/Desktop/python/sample.py
After iteration 1: [34, 25, 12, 22, 11, 64, 90]
After iteration 2: [25, 12, 22, 11, 34, 64, 90]
After iteration 3: [12, 22, 11, 25, 34, 64, 90]
After iteration 4: [12, 11, 22, 25, 34, 64, 90]
After iteration 5: [11, 12, 22, 25, 34, 64, 90]
After iteration 6: [11, 12, 22, 25, 34, 64, 90]
PS C:\Users\User\Desktop\python_ > []
```

3. Write a Python program to implement the concept of insertion sort on the above data set. Print the data set after every iteration.

**Code:**

```
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1

        while j >= 0 and arr[j] > key:
            arr[j + 1] = arr[j]
            j -= 1

        arr[j + 1] = key

        print(f"After iteration {i}: {arr}")

data = [27, 15, 39, 21, 28, 70]
```

```
insertion_sort(data)
```

### Output:

```
PS C:\Users\User\Desktop\python_ > & C:/Users/User/anaconda3/python.exe c:/Users/User/Desktop/python/sample.py
After iteration 1: [15, 27, 39, 21, 28, 70]
After iteration 2: [15, 27, 39, 21, 28, 70]
After iteration 3: [15, 21, 27, 39, 28, 70]
After iteration 4: [15, 21, 27, 28, 39, 70]
After iteration 5: [15, 21, 27, 28, 39, 70]
PS C:\Users\User\Desktop\python_ > █
```

4. Write a Python program to implement the concept of quick sort on the above data set. Print the data set after every iteration.

### Code:

```
def quick_sort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)

        print(f"After partitioning with pivot at index {pi}: {arr}")

        quick_sort(arr, low, pi - 1)
        quick_sort(arr, pi + 1, high)
def partition(arr, low, high):
    pivot = arr[high]
    i = low - 1

    for j in range(low, high):
        if arr[j] < pivot:
            i += 1
            arr[i], arr[j] = arr[j], arr[i]

    arr[i + 1], arr[high] = arr[high], arr[i + 1]

    return i + 1

data_set = [27, 15, 39, 21, 28, 70]
quick_sort(data_set, 0, len(data_set) - 1)
```

### Output:

```
PS C:\Users\User\Desktop\python_ > & C:/Users/User/anaconda3/python.exe c:/Users/User/Desktop/python/sample.py
After partitioning with pivot at index 5: [27, 15, 39, 21, 28, 70]
After partitioning with pivot at index 3: [27, 15, 21, 28, 39, 70]
After partitioning with pivot at index 1: [15, 21, 27, 28, 39, 70]
PS C:\Users\User\Desktop\python_ > █
```

5. Write a Python program to implement the concept of merge sort on the above data set. Print the data set after every iteration.

**Code:**

```
def merge_sort(arr):
    if len(arr) > 1:

        mid = len(arr) // 2

        left_half = arr[:mid]
        right_half = arr[mid:]

        merge_sort(left_half)
        merge_sort(right_half)

        i = j = k = 0

        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1

        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1

        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1

        print(f"After merging: {arr}")

data_set = [27, 15, 39, 21, 28, 70]
merge_sort(data_set)
```

**Output:**

```
PS C:\Users\User\Desktop\python_ > & C:/Users/User/anaconda3/python.exe c:/Users/User/Desktop/python/sample.py
After merging: [15, 39]
After merging: [15, 27, 39]
After merging: [28, 70]
After merging: [21, 28, 70]
After merging: [15, 21, 27, 28, 39, 70]
PS C:\Users\User\Desktop\python_ > █
```

6. Write a Python program to show that Quick sort is better than Bubble sort.

**Code:**

```
import random

def bubble_sort(arr):
    n = len(arr)
    counter = 0
    for i in range(n):
        for j in range(0, n-i-1):
            counter += 1
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
    return counter

def quick_sort(arr):
    counter = 0
    def partition(low, high):
        nonlocal counter
        pivot = arr[high]
        i = low - 1
        for j in range(low, high):
            counter += 1
            if arr[j] <= pivot:
                i += 1
                arr[i], arr[j] = arr[j], arr[i]
        arr[i+1], arr[high] = arr[high], arr[i+1]
        return i + 1

    def quick_sort_recursive(low, high):
        if low < high:
            pi = partition(low, high)
            quick_sort_recursive(low, pi - 1)
            quick_sort_recursive(pi + 1, high)
    quick_sort_recursive(0, len(arr) - 1)
    return counter

def compare_sorts():
    arr_bubble = random.sample(range(1, 1001), 100)
    arr_quick = arr_bubble.copy()
```

```

bubble_iterations = bubble_sort(arr_bubble)
quick_iterations = quick_sort(arr_quick)

print(f"Bubble Sort iterations: {bubble_iterations}")
print(f"Quick Sort iterations: {quick_iterations}")

if bubble_iterations > quick_iterations:
    print("Quick Sort is better.")
elif bubble_iterations < quick_iterations:
    print("Bubble Sort is better.")
else:
    print("Both sorts performed equally well.")

compare_sorts()

```

### Output:

```

Bubble Sort iterations: 4950
Quick Sort iterations: 691
Quick Sort is better.

C:\Users\User\Desktop\Python_eval>

```

7. Write a Python program to show that merge sort is more effective than quick sort.

### Code:

```

import random

def merge_sort(arr):
    counter = 0
    def merge(left, right):
        nonlocal counter
        result = []
        while left and right:
            counter += 1
            if left[0] <= right[0]:
                result.append(left.pop(0))
            else:
                result.append(right.pop(0))
        result += left
        result += right
        return result

    def merge_sort_recursive(arr):
        nonlocal counter
        if len(arr) <= 1:

```

```

        return arr
    mid = len(arr) // 2
    left = merge_sort_recursive(arr[:mid])
    right = merge_sort_recursive(arr[mid:])
    return merge(left, right)

merge_sort_recursive(arr)
return counter

def quick_sort(arr):
    counter = 0
    def partition(low, high):
        nonlocal counter
        pivot = arr[high]
        i = low - 1
        for j in range(low, high):
            counter += 1
            if arr[j] <= pivot:
                i += 1
                arr[i], arr[j] = arr[j], arr[i]
        arr[i+1], arr[high] = arr[high], arr[i+1]
        return i + 1

    def quick_sort_recursive(low, high):
        if low < high:
            pi = partition(low, high)
            quick_sort_recursive(low, pi - 1)
            quick_sort_recursive(pi + 1, high)

    quick_sort_recursive(0, len(arr) - 1)
    return counter

def compare_sorts():
    arr_merge = random.sample(range(1, 1001), 100)
    arr_quick = arr_merge.copy()

    merge_iterations = merge_sort(arr_merge)
    quick_iterations = quick_sort(arr_quick)

    print(f"Merge Sort iterations: {merge_iterations}")
    print(f"Quick Sort iterations: {quick_iterations}")

    if merge_iterations < quick_iterations:
        print("Merge Sort is better.")
    elif merge_iterations > quick_iterations:
        print("Quick Sort is better.")
    else:
        print("Both sorts performed equally well.")

```

```
compare_sorts()
```

**Output:**

```
Merge Sort iterations: 537  
Quick Sort iterations: 590  
Merge Sort is better.  
  
C:\Users\User\Desktop\Python_eval>
```



1. Write a python program to create a binary tree using recursive function and display that level wise.

Code:

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

def insert(root, value):
    if root is None:
        return Node(value)
    else:
        if value < root.value:
            root.left = insert(root.left, value)
        else:
            root.right = insert(root.right, value)
    return root

def level_order_traversal(root):
    if root is None:
        return

    queue = [root]
    while queue:
        current = queue.pop(0)
        print(current.value, end=" ")

        if current.left:
            queue.append(current.left)
        if current.right:
            queue.append(current.right)

def main():
    root = None
    values = [10, 5, 15, 3, 7, 12, 18]

    for value in values:
        root = insert(root, value)
```

```

    print("Level-wise traversal of the binary tree:")
    level_order_traversal(root)

if __name__ == "__main__":
    main()

```

**Output:**

```

PS C:\Users\User> & C:/Users/User/AppData/Local/Programs/Python/Python38/python.exe c:/Users/User/Desktop/As2_1.py
Level-wise traversal of the binary tree:
10 5 15 3 7 12 18

```

2. Write a python program to create a binary tree using non recursive function and display that level wise.

**Code:**

```

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

def insert(root, value):
    new_node = Node(value)

    if root is None:
        return new_node

    queue = [root]

    while queue:
        current = queue.pop(0)

        if current.left is None:
            current.left = new_node
            break
        else:
            queue.append(current.left)

        if current.right is None:
            current.right = new_node
            break
        else:
            queue.append(current.right)

    return root

def level_order_traversal(root):
    if root is None:
        return

    queue = [root]
    while queue:
        current = queue.pop(0)
        print(current.value, end=" ")

        if current.left:
            queue.append(current.left)

```

```

        if current.right:
            queue.append(current.right)

    root = None
    values = [10, 5, 15, 3, 7, 12, 18]

    for value in values:
        root = insert(root, value)

    print("Level-wise traversal of the binary tree:")
    level_order_traversal(root)

if __name__ == "__main__":
    main()

```

**Output:**

```

PS C:\Users\User> & C:/Users/User/AppData/Local/Programs/Python/Python38/python.exe c:/Users/User/Desktop/As2_2.py
Level-wise traversal of the binary tree:
10 5 15 3 7 12 18

```

3. Write a python program to create a binary tree using array only and display that level wise.

**Code:**

```

class BinaryTree:
    def __init__(self, elements):
        self.tree = elements

    def level_order_traversal(self):
        if not self.tree:
            return

        n = len(self.tree)
        for i in range(n):
            print(self.tree[i], end=" ")

def main():

    elements = [10, 5, 15, 3, 7, 12, 18]

    binary_tree = BinaryTree(elements)

    print("Level-wise traversal of the binary tree:")
    binary_tree.level_order_traversal()

if __name__ == "__main__":
    main()

```

**Output:**

```

PS C:\Users\User> & C:/Users/User/AppData/Local/Programs/Python/Python38/python.exe c:/Users/User/Desktop/As2_3.py
Level-wise traversal of the binary tree:
10 5 15 3 7 12 18

```

4. Write a python program to identify the height of a binary tree.

**Code:**

```

class Node:
    def __init__(self, value):

```

```

        self.value = value
        self.left = None
        self.right = None

def calculate_height(root):
    if root is None:
        return -1

    left_height = calculate_height(root.left)
    right_height = calculate_height(root.right)

    return max(left_height, right_height) + 1

def insert(root, value):
    if root is None:
        return Node(value)

    if value < root.value:
        root.left = insert(root.left, value)
    else:
        root.right = insert(root.right, value)

    return root

def main():

    root = None
    values = [10, 5, 15, 3, 7, 12, 18]

    for value in values:
        root = insert(root, value)

    height = calculate_height(root)
    print(f"The height of the binary tree is: {height}")

if __name__ == "__main__":
    main()

```

#### Output:

```

PS C:\Users\User> & C:/Users/User/AppData/Local/Programs/Python/Python38/python.exe c:/Users/User/Desktop/As2_4.py
The height of the binary tree is: 2

```

5. Write a python program to identify degree of a given node.

#### Code:

```

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

def insert(root, value):
    if root is None:
        return Node(value)

    if value < root.value:

```

```

        root.left = insert(root.left, value)
    else:
        root.right = insert(root.right, value)

    return root

def find_degree(root, node_value):
    if root is None:
        return None

    if root.value == node_value:
        degree = 0
        if root.left:
            degree += 1
        if root.right:
            degree += 1
        return degree

    left_degree = find_degree(root.left, node_value)
    if left_degree is not None:
        return left_degree

    right_degree = find_degree(root.right, node_value)
    return right_degree

def main():

    root = None
    values = [10, 5, 15, 3, 7, 12, 18]

    for value in values:
        root = insert(root, value)

    node_value = 5

    degree = find_degree(root, node_value)
    if degree is not None:
        print(f"The degree of the node with value {node_value} is: {degree}")
    else:
        print(f"Node with value {node_value} not found in the tree.")

if __name__ == "__main__":
    main()

```

**Output:**

```

PS C:\Users\User> & C:/Users/User/AppData/Local/Programs/Python/Python38/python.exe c:/Users/User/Desktop/As2_5.py
The degree of the node with value 5 is: 2

```

6. Write a program to count number of leaf node present in a binary tree.

**Code:**

```

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

```

```

def count_leaf_nodes(root):
    if root is None:
        return 0

    if root.left is None and root.right is None:
        return 1

    return count_leaf_nodes(root.left) + count_leaf_nodes(root.right)

def insert(root, value):
    if root is None:
        return Node(value)

    if value < root.value:
        root.left = insert(root.left, value)
    else:
        root.right = insert(root.right, value)

    return root

def main():

    root = None
    values = [10, 5, 15, 3, 7, 12, 18]

    for value in values:
        root = insert(root, value)

    leaf_count = count_leaf_nodes(root)
    print(f"The number of leaf nodes in the binary tree is: {leaf_count}")

if __name__ == "__main__":
    main()

```

**Output:**

```

PS C:\Users\User> & C:/Users/User/AppData/Local/Programs/Python/Python38/python.exe c:/Users/User/Desktop/As2_6.py
The number of leaf nodes in the binary tree is: 4

```

7. Write a program to count number of internal node present in a binary tree.

**Code:**

```

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

def count_leaf_nodes(root):
    if root is None:
        return 0

    if root.left is None and root.right is None:
        return 1

    return count_leaf_nodes(root.left) + count_leaf_nodes(root.right)

```

```

def insert(root, value):
    if root is None:
        return Node(value)

    if value < root.value:
        root.left = insert(root.left, value)
    else:
        root.right = insert(root.right, value)

    return root

def main():

    root = None
    values = [10, 5, 15, 3, 7, 12, 18]

    for value in values:
        root = insert(root, value)

    leaf_count = count_leaf_nodes(root)
    print(f"The number of leaf nodes in the binary tree is: {leaf_count}")

if __name__ == "__main__":
    main()

```

**Output:**

```

PS C:\Users\User> & C:/Users/User/AppData/Local/Programs/Python/Python38/python.exe c:/Users/User/Desktop/As2_7.py
The number of internal nodes in the binary tree is: 3

```

8. Write a program to count number of node present in a given binary tree using linked list.

**Code:**

```

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

def count_nodes(root):
    if root is None:
        return 0

    return 1 + count_nodes(root.left) + count_nodes(root.right)

def insert(root, value):
    if root is None:
        return Node(value)

    if value < root.value:
        root.left = insert(root.left, value)
    else:
        root.right = insert(root.right, value)

    return root

```

```
def main():

    root = None
    values = [10, 5, 15, 3, 7, 12, 18]

    for value in values:
        root = insert(root, value)

    total_nodes = count_nodes(root)
    print(f"The total number of nodes in the binary tree is: {total_nodes}")

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

**Output:**

```
PS C:\Users\User> & C:/Users/User/AppData/Local/Programs/Python/Python38/python.exe c:/Users/User/Desktop/As2_8.py
The total number of nodes in the binary tree is: 7
```

9. Write a program to count number of node present in a given binary tree using array.

**Code:**

```
class BinaryTree:
    def __init__(self):
        self.tree = []

    def insert(self, value):
        self.tree.append(value)

    def count_nodes(self):
        return len(self.tree)

def main():

    binary_tree = BinaryTree()

    values = [10, 5, 15, 3, 7, 12, 18]
    for value in values:
        binary_tree.insert(value)

    total_nodes = binary_tree.count_nodes()
    print(f"The total number of nodes in the binary tree is: {total_nodes}")

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

**Output:**

```
PS C:\Users\User> & C:/Users/User/AppData/Local/Programs/Python/Python38/python.exe c:/Users/User/Desktop/As2_9.py
The total number of nodes in the binary tree is: 7
```

10. Write a program to count number of siblings present in a binary tree.

**Code:**

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



```

        self.right = None

def count_siblings(root):
    if root is None:
        return 0

    siblings_count = 0

    if root.left is not None and root.right is not None:
        siblings_count += 2

    elif root.left is not None or root.right is not None:
        siblings_count += 1

    siblings_count += count_siblings(root.left) + count_siblings(root.right)

    return siblings_count

def insert(root, value):
    if root is None:
        return Node(value)

    if value < root.value:
        root.left = insert(root.left, value)
    else:
        root.right = insert(root.right, value)

    return root

def main():
    root = None

    values = [10, 5, 15, 3, 7, 12, 18]

    for value in values:
        root = insert(root, value)

    total_siblings = count_siblings(root)
    print(f"The total number of siblings in the binary tree is: {total_siblings}")

if __name__ == "__main__":
    main()

```

**Output:**

```

PS C:\Users\User> & C:/Users/User/AppData/Local/Programs/Python/Python38/python.exe c:/Users/User/Desktop/As2_10.py
The total number of siblings in the binary tree is: 6

```

## Assignment 3

1. Write a Python program to create a binary search tree using recursive function and display that.

Code:

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

def insert(root, key):
    if root is None:
        return Node(key)
    else:
        if key < root.value:
            root.left = insert(root.left, key)
        else:
            root.right = insert(root.right, key)
    return root

def inorder(root):
    if root:
        inorder(root.left)
        print(root.value, end=' ')
        inorder(root.right)

root = None
n = int(input("Enter number of elements to insert in the BST: "))
for _ in range(n):
    value = int(input("Enter a value to insert: "))
    root = insert(root, value)

print("Inorder Traversal of the BST:")
inorder(root)
```

**Output:**

```
Enter number of elements to insert in the BST: 5
Enter a value to insert: 5
Enter a value to insert: 11
Enter a value to insert: 40
Enter a value to insert: 62
Enter a value to insert: 78
Inorder Traversal of the BST:
5 11 40 62 78
```

2. Write a Python program to create a binary search tree using non-recursive function and display that.

**Code:**

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

def insert_iterative(root, key):
    new_node = Node(key)
    if root is None:
        return new_node
    current = root
    while True:
        if key < current.value:
            if current.left is None:
                current.left = new_node
                break
            current = current.left
        else:
            if current.right is None:
                current.right = new_node
                break
            current = current.right
    return root

def inorder(root):
    if root:
        inorder(root.left)
        print(root.value, end=' ')
        inorder(root.right)

root = None
n = int(input("Enter number of elements to insert in the BST: "))
for _ in range(n):
    value = int(input("Enter a value to insert: "))
    root = insert_iterative(root, value)

print("Inorder Traversal of the BST:")
```

```
inorder(root)
```

**Output:**

```
Enter number of elements to insert in the BST: 5
Enter a value to insert: 8
Enter a value to insert: 25
Enter a value to insert: 74
Enter a value to insert: 30
Enter a value to insert: 21
Inorder Traversal of the BST:
8 21 25 30 74
```

**3. Write a Python program to insert (by using a function) a specific element into an existing binary search tree and then display that.**

**Code:**

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

def insert(root, key):
    if root is None:
        return Node(key)
    else:
        if key < root.value:
            root.left = insert(root.left, key)
        else:
            root.right = insert(root.right, key)
    return root

def inorder(root):
    if root:
        inorder(root.left)
        print(root.value, end=' ')
        inorder(root.right)

root = None
n = int(input("Enter number of elements to insert in the BST: "))
for _ in range(n):
    value = int(input("Enter a value to insert: "))
    root = insert(root, value)

new_value = int(input("Enter the value to insert into the BST: "))
root = insert(root, new_value)
```

```
print("Inorder Traversal of the BST after insertion:")
inorder(root)
```

**Output:**

```
Enter number of elements to insert in the BST: 6
Enter a value to insert: 40
Enter a value to insert: 2
Enter a value to insert: 98
Enter a value to insert: 30
Enter a value to insert: 5
Enter a value to insert: 25
Enter the value to insert into the BST: 45
Inorder Traversal of the BST after insertion:
2 5 25 30 40 45 98
```

**4. Write a Python program to search an element in a BST and show the result.**

**Code:**

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

def insert(root, key):
    if root is None:
        return Node(key)
    else:
        if key < root.value:
            root.left = insert(root.left, key)
        else:
            root.right = insert(root.right, key)
    return root

def search(root, key):
    if root is None or root.value == key:
        return root
    if key > root.value:
        return search(root.right, key)
    return search(root.left, key)

root = None
n = int(input("Enter number of elements to insert in the BST: "))
for _ in range(n):
    value = int(input("Enter a value to insert: "))
    root = insert(root, value)

key = int(input("Enter the value to search in the BST: "))
```

```

result = search(root, key)

if result:
    print(f"Element {key} found in the BST.")
else:
    print(f"Element {key} not found in the BST.")

```

### Output:

```

Enter number of elements to insert in the BST: 5
Enter a value to insert: 78
Enter a value to insert: 5
Enter a value to insert: 2
Enter a value to insert: 93
Enter a value to insert: 70
Enter the value to search in the BST: 70
Element 70 found in the BST.

```

5. Write a Python program to take user name as input and display the sorted sequence of characters using BST.

### Code:

```

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

def insert(root, key):
    if root is None:
        return Node(key)
    else:
        if key < root.value:
            root.left = insert(root.left, key)
        else:
            root.right = insert(root.right, key)
    return root

def inorder(root):
    if root:
        inorder(root.left)
        print(root.value, end=' ')
        inorder(root.right)

name = input("Enter your name: ")
root = None

for char in name:

```

```

    root = insert(root, char)

print("Sorted sequence of characters using BST:")
inorder(root)

```

### Output:

```

Enter your name: Ritu Chatterjee
Sorted sequence of characters using BST:
C R a e e e h i j r t t t u

```

6. Write a Python program to sort a given set of integers using BST.

### Code:

```

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

def insert(root, key):
    if root is None:
        return Node(key)
    else:
        if key < root.value:
            root.left = insert(root.left, key)
        else:
            root.right = insert(root.right, key)
    return root

def inorder(root):
    if root:
        inorder(root.left)
        print(root.value, end=' ')
        inorder(root.right)

nums = list(map(int, input("Enter a set of integers to sort (comma separated):").split(',')))
root = None

for num in nums:
    root = insert(root, num)

print("Sorted sequence using BST:")
inorder(root)

```

### Output:

```
Enter a set of integers to sort: 58,64,21,48,98,25
Sorted sequence using BST:
21 25 48 58 64 98
PS C:\Users\User\Desktop\MY Folder> |
```

7. Write a Python program to display a BST using In-order, Pre-order, Post-order.

Code:

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

def insert(root, key):
    if root is None:
        return Node(key)
    else:
        if key < root.value:
            root.left = insert(root.left, key)
        else:
            root.right = insert(root.right, key)
    return root

def inorder(root):
    if root:
        inorder(root.left)
        print(root.value, end=' ')
        inorder(root.right)

def preorder(root):
    if root:
        print(root.value, end=' ')
        preorder(root.left)
        preorder(root.right)

def postorder(root):
    if root:
        postorder(root.left)
        postorder(root.right)
        print(root.value, end=' ')

root = None

n = int(input("Enter number of elements to insert in the BST: "))
for _ in range(n):
    value = int(input("Enter a value to insert: "))
    root = insert(root, value)

print("Inorder Traversal of the BST:")
inorder(root)
```



```

print("\nPreorder Traversal of the BST:")
preorder(root)
print("\nPostorder Traversal of the BST:")
postorder(root)

```

### Output:

```

Enter number of elements to insert in the BST: 5
Enter a value to insert: 54
Enter a value to insert: 82
Enter a value to insert: 70
Enter a value to insert: 32
Enter a value to insert: 5
Inorder Traversal of the BST:
5 32 54 70 82
Preorder Traversal of the BST:
54 32 5 82 70
Postorder Traversal of the BST:
5 32 70 82 54

```

8. Write a Python program to Count the number of nodes present in an existing BST and display the highest element present in the BST.

### Code:

```

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

def insert(root, key):
    if root is None:
        return Node(key)
    else:
        if key < root.value:
            root.left = insert(root.left, key)
        else:
            root.right = insert(root.right, key)
    return root

def count_nodes(root):
    if root is None:
        return 0
    return 1 + count_nodes(root.left) + count_nodes(root.right)

def find_max(root):
    current = root
    while current.right:
        current = current.right
    return current.value

```

```

root = None

n = int(input("Enter number of elements to insert in the BST: "))
for _ in range(n):
    value = int(input("Enter a value to insert: "))
    root = insert(root, value)

node_count = count_nodes(root)
max_value = find_max(root)

print(f"Total number of nodes in the BST: {node_count}")
print(f"Highest element in the BST: {max_value}")

```

### Output:

```

Enter number of elements to insert in the BST: 5
Enter a value to insert: 54
Enter a value to insert: 87
Enter a value to insert: 25
Enter a value to insert: 108
Enter a value to insert: 30
Total number of nodes in the BST: 5
Highest element in the BST: 108

```

9. Write a Python program to prove that binary search tree is better than binary tree.

### Code:

```

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

class BST:
    def __init__(self):
        self.root = None
        self.comparisons = 0

    def insert(self, key):
        new_node = Node(key)
        if self.root is None:
            self.root = new_node
            return
        current = self.root
        while True:
            if key < current.val:
                if current.left is None:
                    current.left = new_node
                    return
            else:
                if current.right is None:
                    current.right = new_node
                    return
            current = current.left or current.right

```

```

        current = current.left
    else:
        if current.right is None:
            current.right = new_node
            return
        current = current.right

def search(self, node, key):
    if node is None:
        return None
    self.comparisons += 1
    if node.val == key:
        return node
    elif key < node.val:
        return self.search(node.left, key)
    else:
        return self.search(node.right, key)

def build_balanced_bst(self, sorted_elements):
    self.root = self._build_balanced_bst_helper(sorted_elements, 0,
len(sorted_elements) - 1)

def _build_balanced_bst_helper(self, sorted_elements, start, end):
    if start > end:
        return None
    mid = (start + end) // 2
    node = Node(sorted_elements[mid])
    node.left = self._build_balanced_bst_helper(sorted_elements, start,
mid - 1)
    node.right = self._build_balanced_bst_helper(sorted_elements, mid + 1,
end)
    return node

class BinaryTree:
    def __init__(self):
        self.nodes = []
        self.comparisons = 0

    def insert(self, key):
        self.nodes.append(key)

    def search(self, key):
        for node in self.nodes:
            self.comparisons += 1
            if node == key:
                return node
        return None

bst = BST()
bt = BinaryTree()

```

```
elements = list(map(int, input("Enter elements for BST and Binary Tree: ").split()))

sorted_elements = sorted(elements)
bst.build_balanced_bst(sorted_elements)

for num in elements:
    bt.insert(num)

search_key = int(input("Enter an element to search in the BST and Binary Tree: "))

bst.search(bst.root, search_key)
bt.search(search_key)

print(f"Number of iterations in BST: {bst.comparisons}")
print(f"Number of iterations in Binary Tree: {bt.comparisons}")
```

**Output:**

```
Enter elements for BST and Binary Tree: 15 16 17 18 19
Enter an element to search in the BST and Binary Tree: 18
Number of iterations in BST: 2
Number of iterations in Binary Tree: 4
```

1. Write a Python program to search an element recursively in a binary search tree.

Code:

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

def search(root, key):
    if root is None or root.value == key:
        return root
    if key > root.value:
        return search(root.right, key)
    return search(root.left, key)

def inorder(root):
    if root:
        inorder(root.left)
        print(root.value, end=" ")
        inorder(root.right)

def build_tree():
    root = Node(int(input("Enter root node: ")))
    while True:
        choice = input("Do you want to add a node? (y/n): ")
        if choice == 'y':
            value = int(input("Enter node value: "))
            insert(root, value)
        else:
            break
    return root

def insert(root, value):
    if value < root.value:
        if root.left is None:
```

```

        root.left = Node(value)
    else:
        insert(root.left, value)
    elif value > root.value:
        if root.right is None:
            root.right = Node(value)
        else:
            insert(root.right, value)

root = build_tree()
inorder(root)
print("\n")

key = int(input("Enter the key to search: "))
result = search(root, key)
if result:
    print(f"Element {key} found in the tree.")
else:
    print(f"Element {key} not found in the tree.")

```

**Output:**

```

Enter root node: 50
Do you want to add a node? (y/n): y
Enter node value: 30
Do you want to add a node? (y/n): y
Enter node value: 70
Do you want to add a node? (y/n): n
30 50 70

Enter the key to search: 30
Element 30 found in the tree.

```

2. Write a Python program to delete a child node from a binary search tree.

**Code:**

```

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

def delete_node(root, key):
    if root is None:
        return root
    if key < root.value:
        root.left = delete_node(root.left, key)
    elif key > root.value:
        root.right = delete_node(root.right, key)
    else:
        if root.left is None:

```

```

        temp = root.right
        root = None
        return temp
    elif root.right is None:
        temp = root.left
        root = None
        return temp
    return root

def build_tree():
    root = Node(int(input("Enter root node: ")))
    while True:
        choice = input("Do you want to add a node? (y/n): ")
        if choice == 'y':
            value = int(input("Enter node value: "))
            insert(root, value)
        else:
            break
    return root

def insert(root, value):
    if value < root.value:
        if root.left is None:
            root.left = Node(value)
        else:
            insert(root.left, value)
    elif value > root.value:
        if root.right is None:
            root.right = Node(value)
        else:
            insert(root.right, value)

def inorder(root):
    if root:
        inorder(root.left)
        print(root.value, end=" ")
        inorder(root.right)

root = build_tree()
print("\nTree before deletion: ")
inorder(root)
print("\n")

key = int(input("Enter the node value to delete: "))
root = delete_node(root, key)

print("\nTree after deletion: ")
inorder(root)

```

Output:

```
Enter root node: 50
Do you want to add a node? (y/n): y
Enter node value: 30
Do you want to add a node? (y/n): y
Enter node value: 70
Do you want to add a node? (y/n): n

Tree before deletion:
30 50 70

Enter the node value to delete: 30

Tree after deletion:
50 70
```

3. Write a Python program to delete a node having one child from a binary search tree.

Code:

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

def delete_node_with_one_child(root, key):
    if root is None:
        return root
    if key < root.value:
        root.left = delete_node_with_one_child(root.left, key)
    elif key > root.value:
        root.right = delete_node_with_one_child(root.right, key)
    else:
        if root.left is None:
            temp = root.right
            root = None
            return temp
        elif root.right is None:
            temp = root.left
            root = None
            return temp
    return root

def inorder(root):
    if root:
        inorder(root.left)
        print(root.value, end=" ")
        inorder(root.right)

def build_tree():
    root = Node(int(input("Enter root node: ")))
```



```

while True:
    choice = input("Do you want to add a node? (y/n): ")
    if choice == 'y':
        value = int(input("Enter node value: "))
        insert(root, value)
    else:
        break
return root

def insert(root, value):
    if value < root.value:
        if root.left is None:
            root.left = Node(value)
        else:
            insert(root.left, value)
    elif value > root.value:
        if root.right is None:
            root.right = Node(value)
        else:
            insert(root.right, value)

root = build_tree()
print("\nTree before deletion: ")
inorder(root)
print("\n")

key = int(input("Enter the node value to delete (node with one child): "))
root = delete_node_with_one_child(root, key)

print("\nTree after deletion: ")
inorder(root)

```

**Output:**

```

Enter root node: 50
Do you want to add a node? (y/n): y
Enter node value: 30
Do you want to add a node? (y/n): y
Enter node value: 70
Do you want to add a node? (y/n): y
Enter node value: 40
Do you want to add a node? (y/n): n

Tree before deletion:
30 40 50 70

Enter the node value to delete (node with one child): 40

Tree after deletion:
30 50 70

```

4. Write a Python program to delete a node having two children from a binary search tree.

Code:

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

def min_value_node(root):
    current = root
    while current.left is not None:
        current = current.left
    return current

def delete_node_with_two_children(root, key):
    if root is None:
        return root
    if key < root.value:
        root.left = delete_node_with_two_children(root.left, key)
    elif key > root.value:
        root.right = delete_node_with_two_children(root.right, key)
    else:
        if root.left is not None and root.right is not None:
            temp = min_value_node(root.right)
            root.value = temp.value
            root.right = delete_node_with_two_children(root.right,
temp.value)
        elif root.left is None:
            temp = root.right
            root = None
            return temp
        elif root.right is None:
            temp = root.left
            root = None
            return temp
    return root

def inorder(root):
    if root:
        inorder(root.left)
        print(root.value, end=" ")
        inorder(root.right)

def build_tree():
    root = Node(int(input("Enter root node: ")))
    while True:
        choice = input("Do you want to add a node? (y/n): ")
        if choice == 'y':
            value = int(input("Enter node value: "))
```

```

        insert(root, value)
    else:
        break
    return root

def insert(root, value):
    if value < root.value:
        if root.left is None:
            root.left = Node(value)
        else:
            insert(root.left, value)
    elif value > root.value:
        if root.right is None:
            root.right = Node(value)
        else:
            insert(root.right, value)

# Main program
root = build_tree()
print("\nTree before deletion: ")
inorder(root)
print("\n")

key = int(input("Enter the node value to delete (node with two
children): "))
root = delete_node_with_two_children(root, key)

print("\nTree after deletion: ")
inorder(root)

```

**Output:**

```

Enter root node: 50
Do you want to add a node? (y/n): y
Enter node value: 30
Do you want to add a node? (y/n): y
Enter node value: 70
Do you want to add a node? (y/n): y
Enter node value: 40
Do you want to add a node? (y/n): y
Enter node value: 60
Do you want to add a node? (y/n): n

Tree before deletion:
30 40 50 60 70

Enter the node value to delete (node with two children): 30

Tree after deletion:
40 50 60 70

```

5. Write a Python program to delete a node from a binary search tree.

Code:

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

def delete_node(root, key):
    if root is None:
        return root
    if key < root.value:
        root.left = delete_node(root.left, key)
    elif key > root.value:
        root.right = delete_node(root.right, key)
    else:
        if root.left is None:
            temp = root.right
            root = None
            return temp
        elif root.right is None:
            temp = root.left
            root = None
            return temp
        temp = min_value_node(root.right)
        root.value = temp.value
        root.right = delete_node(root.right, temp.value)
    return root

def min_value_node(root):
    current = root
    while current.left is not None:
        current = current.left
    return current

def inorder(root):
    if root:
        inorder(root.left)
        print(root.value, end=" ")
        inorder(root.right)

def build_tree():
    root = Node(int(input("Enter root node: ")))
    while True:
        choice = input("Do you want to add a node? (y/n): ")
        if choice == 'y':
            value = int(input("Enter node value: "))
            insert(root, value)
        else:
```

```

        break
    return root

def insert(root, value):
    if value < root.value:
        if root.left is None:
            root.left = Node(value)
        else:
            insert(root.left, value)
    elif value > root.value:
        if root.right is None:
            root.right = Node(value)
        else:
            insert(root.right, value)

root = build_tree()
print("\nTree before deletion: ")
inorder(root)
print("\n")

key = int(input("Enter the node value to delete: "))
root = delete_node(root, key)

print("\nTree after deletion: ")
inorder(root)

```

#### Output:

```

Enter root node: 50
Do you want to add a node? (y/n): y
Enter node value: 30
Do you want to add a node? (y/n): y
Enter node value: 70
Do you want to add a node? (y/n): y
Enter node value: 40
Do you want to add a node? (y/n): n

Tree before deletion:
30 40 50 70

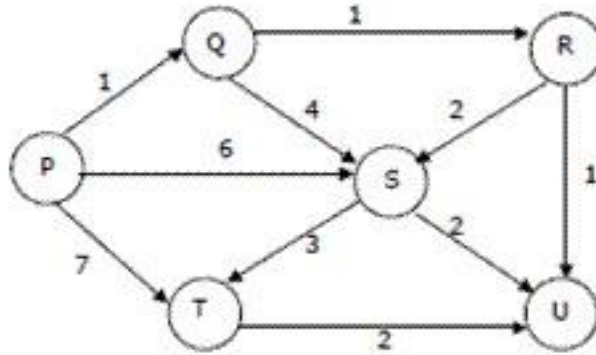
Enter the node value to delete: 40

Tree after deletion:
30 50 70

```

## Topic: To implement Data Structure Graph and relevant properties

1. Write a Python program to store the following Graph using Adjacency Matrix & display that.



---

### Code:

```
num_vertices = 6
```

```
vertex_index = {'P': 0, 'Q': 1, 'R': 2, 'S': 3, 'T': 4, 'U': 5}
```

```
adj_matrix = [[0 for _ in range(num_vertices)] for _ in range(num_vertices)]
```

```
edges = {
```

```
    ('P', 'Q'): 1,
```

```
    ('P', 'S'): 6,
```

```
    ('P', 'T'): 7,
```

```
    ('Q', 'R'): 1,
```

```
    ('Q', 'S'): 4,
```

```
    ('R', 'S'): 2,
```

```
    ('S', 'T'): 3,
```

```
    ('S', 'U'): 2,
```

```
    ('R', 'U'): 1,
```

```
    ('T', 'U'): 2
```

```
}
```

```

for (node1, node2), weight in edges.items():
    i, j = vertex_index[node1], vertex_index[node2]
    adj_matrix[i][j] = weight

```

```

print("Adjacency Matrix:")
for row in adj_matrix:
    print(row)

```

```

G = nx.DiGraph()

```

```

G.add_nodes_from(vertex_index.keys())

```

```

for (node1, node2), weight in edges.items():
    G.add_edge(node1, node2, weight=weight)

```

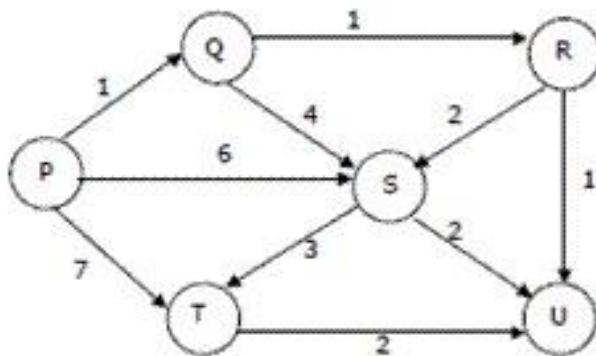
**Output:**

```

➡ Adjacency Matrix:
[0, 1, 0, 6, 7, 0]
[0, 0, 1, 4, 0, 0]
[0, 0, 0, 2, 0, 1]
[0, 0, 0, 0, 3, 2]
[0, 0, 0, 0, 0, 2]
[0, 0, 0, 0, 0, 0]

```

2. Write a Python program to store the following Graph using Adjacency List & display that.



**Code:**

```

num_vertices = 6

```

```

vertex_index = {'P': 0, 'Q': 1, 'R': 2, 'S': 3, 'T': 4, 'U': 5}

```

```

adj_list = {node: [] for node in vertex_index.keys()}

```

```
edges = {
    ('P', 'Q'): 1,
    ('P', 'S'): 6,
    ('P', 'T'): 7,
    ('Q', 'R'): 1,
    ('Q', 'S'): 4,
    ('R', 'S'): 2,
    ('S', 'T'): 3,
    ('S', 'U'): 2,
    ('R', 'U'): 1,
    ('T', 'U'): 2
}
```

```
for (node1, node2), weight in edges.items():
    adj_list[node1].append((node2, weight))
```

```
print("Adjacency List:")
for node, neighbors in adj_list.items():
    print(f"{node}: {neighbors}")
```

```
G = nx.DiGraph()
```

```
G.add_nodes_from(vertex_index.keys())
```

```
for (node1, node2), weight in edges.items():
    G.add_edge(node1, node2, weight=weight)
```

### Output:

```
➡ Adjacency List:
P: [('Q', 1), ('S', 6), ('T', 7)]
Q: [('R', 1), ('S', 4)]
R: [('S', 2), ('U', 1)]
S: [('T', 3), ('U', 2)]
T: [('U', 2)]
U: []
```

3. Write a Python program to count number of vertices and edges present in a graph.

### Code:

```
class GraphList:
    def __init__(self):
        self.graph = {}
```



```
def add_edge(self, u, v, directed=False):
```

```
    if u not in self.graph:
```

```
        self.graph[u] = []
```

```
    if v not in self.graph:
```

```
        self.graph[v] = []
```

```
    self.graph[u].append(v)
```

```
    if not directed:
```

```
        self.graph[v].append(u)
```

```
def count_vertices_edges(graph, directed=False):
```

```
    vertices = len(graph)
```

```
    edges = sum(len(adj) for adj in graph.values())
```

```
    if not directed:
```

```
        edges //= 2
```

```
    return vertices, edges
```

```
g = GraphList()
```

```
g.add_edge(0, 1)
```

```
g.add_edge(0, 4)
```

```
g.add_edge(1, 2)
```

```
g.add_edge(1, 3)
```

```
g.add_edge(1, 4)
```

```
g.add_edge(2, 3)
```

```
g.add_edge(3, 4)
```

```
vertices, edges = count_vertices_edges(g.graph)
```

```
print("Number of Vertices:", vertices)
```

```
print("Number of Edges:", edges)
```

**Output:**

```
↔ Number of Vertices: 5  
   Number of Edges: 7
```

4. Write a Python program to detect a cycle in a graph.

**Code:**

```
class Graph:
```

```
    def __init__(self):
```

```
        self.graph = {}
```

```
    def add_edge(self, u, v, directed=False):
```

```
        if u not in self.graph:
```

```
            self.graph[u] = []
```

```
        if v not in self.graph:
```

```
            self.graph[v] = []
```

```
        self.graph[u].append(v)
```

```
        if not directed:
```

```
            self.graph[v].append(u)
```


```
def has_cycle_util(self, v, visited, parent):
    visited[v] = True
    for neighbor in self.graph[v]:
        if not visited[neighbor]:
            if self.has_cycle_util(neighbor, visited, v):
                return True
        elif neighbor != parent:
            return True
    return False
```

```
def has_cycle(self):
    visited = {node: False for node in self.graph}
    for node in self.graph:
        if not visited[node]:
            if self.has_cycle_util(node, visited, -1):
                return True
    return False
```

```
g = Graph()
g.add_edge(0, 1)
g.add_edge(1, 2)
g.add_edge(2, 0)
g.add_edge(2, 3)
```

```
print("Cycle Detected" if g.has_cycle() else "No Cycle Found")
```

**Output:**

 Cycle Detected

5. Write a Python program to identify number of odd degree vertices and number of even degree vertices in a graph.

**Code:**

```
class Graph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, u, v):
        if u not in self.graph:
            self.graph[u] = []
        if v not in self.graph:
            self.graph[v] = []
        self.graph[u].append(v)
        self.graph[v].append(u)

    def count_odd_even_degrees(self):
        odd_count = even_count = 0
        for node in self.graph:
            degree = len(self.graph[node])
            if degree % 2 == 0:
                even_count += 1
            else:
```


```
        odd_count += 1
    return odd_count, even_count
```

```
g = Graph()
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(1, 2)
g.add_edge(1, 3)
g.add_edge(2, 3)
```

```
odd, even = g.count_odd_even_degrees()
print("Odd Degree Vertices:", odd)
print("Even Degree Vertices:", even)
```

**Output:**

---

 Odd Degree Vertices: 2  
Even Degree Vertices: 2

6. Write a Python program to check whether a given graph is complete or not.

**Code:**

```
class Graph:
    def __init__(self, vertices):
        self.num_vertices = vertices
        self.graph = {i: set() for i in range(vertices)}

    def add_edge(self, u, v):
        self.graph[u].add(v)
        self.graph[v].add(u)


    def is_complete(self):
        for node in self.graph:
            if len(self.graph[node]) != self.num_vertices - 1:
                return False
        return True
```

```
g = Graph(4)
g.add_edge(0, 1)
g.add_edge(0, 2)
g.add_edge(0, 3)
g.add_edge(1, 2)
g.add_edge(1, 3)
g.add_edge(2, 3)
```

```
print("Graph is Completed" if g.is_complete() else "Graph is Not Complete")
```

**Output:**

---

 Graph is Complete