PYTHON USING DATA STRUCTURE PROGRAMMING

Dt:06-06-2024

LINKED-LIST

CODE:

```
#LINKED-LIST
# Create a Node class to create a node
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
# Create a LinkedList class
class LinkedList:
    def __init__(self):
        self.head = None
    # Method to add a node at the beginning of the LL
    def insertAtBegin(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            return
        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):
        new_node = Node(data)
```

```
    current_node = self.head
    position = 0
    if index == 0:
        self.insertAtBegin(data)
        return
    else:
        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.next = current_node.next
            current_node.next = new_node
        else:
            print("Index not present")
# Method to add a node at the end of the 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 update a node of the linked list at a given position
def updateNode(self, val, index):
```

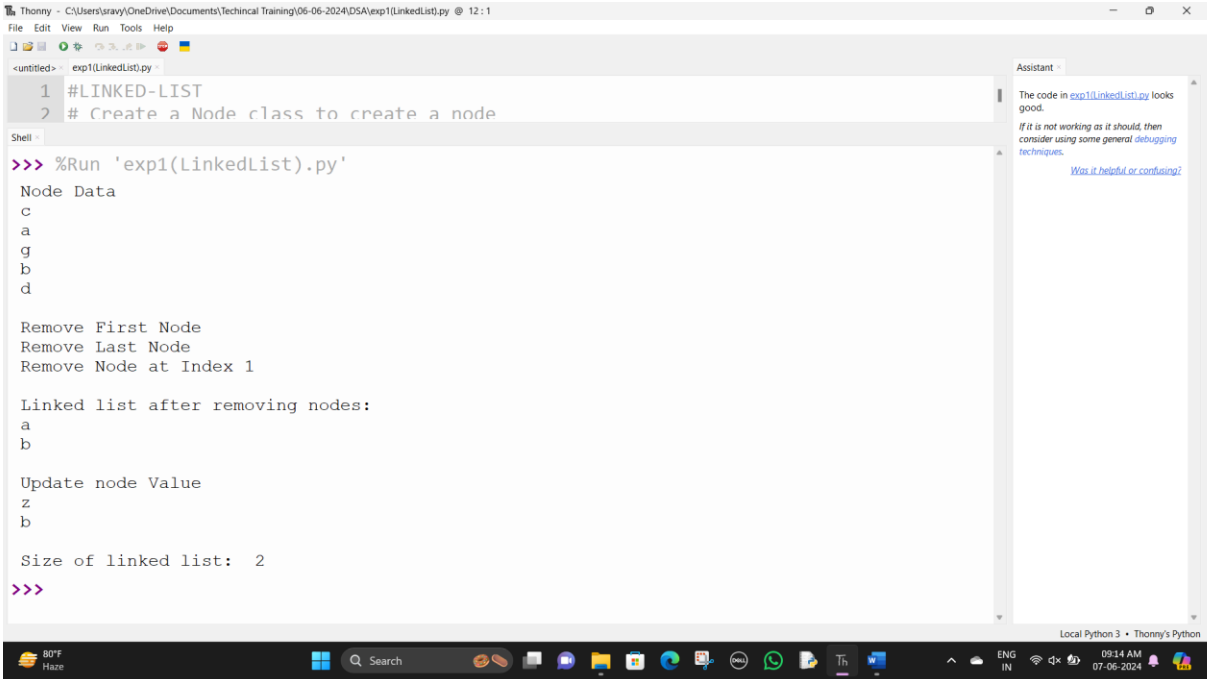
```
    current_node = self.head
    position = 0
    if index == 0:
        if current_node is not None:
            current_node.data = val
        return
    else:
        while current_node is not None and position != index:
            position += 1
            current_node = current_node.next
        if current_node is not None:
            current_node.data = val
        else:
            print("Index not present")
# Method to remove the first node of the linked list
def remove_first_node(self):
    if self.head is None:
        return
    self.head = self.head.next
# Method to remove the last node of the linked list
def remove_last_node(self):
    if self.head is None:
        return
    current_node = self.head
    if current_node.next is None:
        self.head = None
```

```
        return
    while current_node.next.next:
        current_node = current_node.next
    current_node.next = None
# Method to remove a node at a given index
def remove_at_index(self, index):
    if self.head is None:
        return
  current_node = self.head
    position = 0
    if index == 0:
        self.remove_first_node()
        return
    else:
        while current_node is not None and position + 1 != index:
            position += 1
            current_node = current_node.next
        if current_node is not None and current_node.next is not None:
            current_node.next = current_node.next.next
        else:
            print("Index not present")
# Method to remove a node by data value
def remove_node(self, data):
    current_node = self.head
    if current_node is not None and current_node.data == data:
        self.remove_first_node()
```

```
            return
        while current_node is not None and current_node.next is not None and
current_node.next.data != data:
            current_node = current_node.next
        if current_node is None or current_node.next is None:
            return
        else:
            current_node.next = current_node.next.next
    # Method to print the size of the linked list
    def sizeOfLL(self):
        size = 0
        current_node = self.head
        while current_node:
            size += 1
            current_node = current_node.next
        return size
    # Method to print the linked list
    def printLL(self):
        current_node = self.head
        while current_node:
            print(current_node.data)
            current_node = current_node.next
# Create a new linked list
llist = LinkedList()
# Add nodes to the linked list
llist.insertAtEnd('a')
llist.insertAtEnd('b')
```

```
llist.insertAtBegin('c')

llist.insertAtEnd('d')

llist.insertAtIndex('g', 2)

# Print the linked list

print("Node Data")

llist.printLL()

# Remove nodes from the linked list

print("\nRemove First Node")

llist.remove_first_node()

print("Remove Last Node")

llist.remove_last_node()

print("Remove Node at Index 1")

llist.remove_at_index(1)

# Print the linked list again

print("\nLinked list after removing nodes:")

llist.printLL()

print("\nUpdate node Value")

llist.updateNode('z', 0)

llist.printLL()

print("\nSize of linked list: ", end=" ")

print(llist.sizeOfLL())
```

OUTPUT:



<u>DOUBLE LINKED-LIST</u>

CODE:

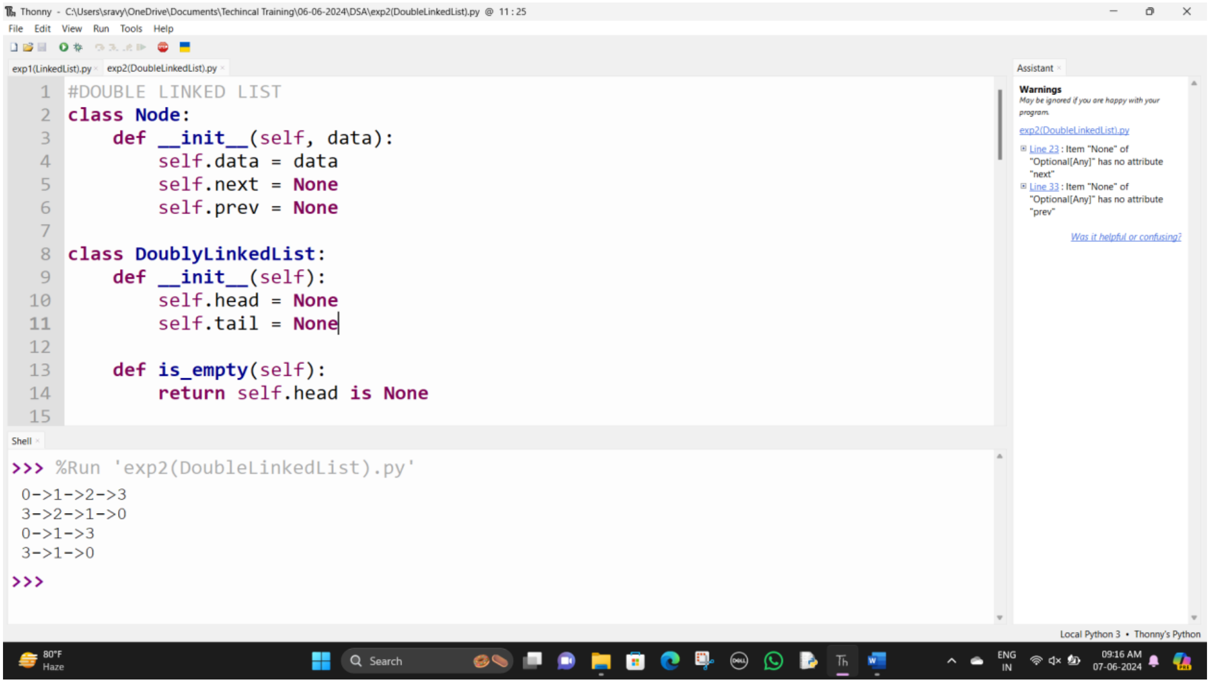
```
#DOUBLE LINKED LIST
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None
class DoublyLinkedList:
    def __init__(self):
        self.head = None
        self.tail = None
    def is_empty(self):
        return self.head is None
    def append(self, data):
```

```
        new_node = Node(data)
        if self.is_empty():
            self.head = new_node
            self.tail = new_node
        else:
            new_node.prev = self.tail
            self.tail.next = new_node
            self.tail = new_node
    def prepend(self, data):
        new_node = Node(data)
        if self.is_empty():
            self.head = new_node
            self.tail = new_node
        else:
            new_node.next = self.head
            self.head.prev = new_node
            self.head = new_node
    def delete(self, data):
        if self.is_empty():
            return
        current = self.head
        while current is not None and current.data != data:
            current = current.next
        if current is not None:
            if current.prev is not None:
                current.prev.next = current.next
```

```
            else:
                self.head = current.next
            if current.next is not None:
                current.next.prev = current.prev
            else:
                self.tail = current.prev
    def display_forward(self):
        elements = []
        current = self.head
        while current:
            elements.append(current.data)
            current = current.next
        print("->".join(map(str, elements)))
    def display_backward(self):
        elements = []
        current = self.tail
        while current:
            elements.append(current.data)
            current = current.prev
        print("->".join(map(str, elements)))
my_doubly_linked_list = DoublyLinkedList()
my_doubly_linked_list.append(1)
my_doubly_linked_list.append(2)
my_doubly_linked_list.append(3)
my_doubly_linked_list.prepend(0)
my_doubly_linked_list.display_forward()
```

```
my_doubly_linked_list.display_backward()

my_doubly_linked_list.delete(2)

my_doubly_linked_list.display_forward()

my_doubly_linked_list.display_backward()
```

OUTPUT:



CIRCULAR LINKED-LIST

CODE:

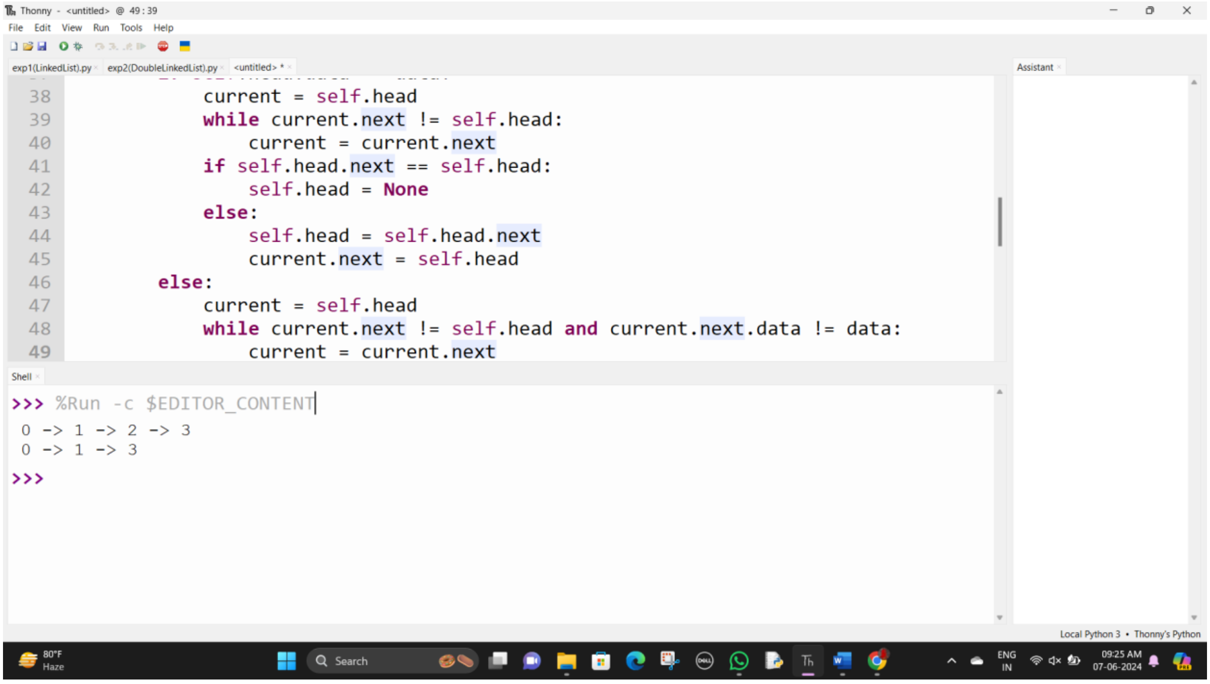
```
#CIRCULAR LINKED-LIST

class Node:

    def __init__(self, data=None):

        self.data = data

        self.next = None

class CircularLinkedList:

    def __init__(self):

        self.head = None

    def is_empty(self):
```

```
    return self.head is None
def append(self, data):
    new_node = Node(data)
    if self.is_empty():
        self.head = new_node
        new_node.next = self.head
    else:
        current = self.head
        while current.next != self.head:
            current = current.next
        current.next = new_node
        new_node.next = self.head
def prepend(self, data):
    new_node = Node(data)
    if self.is_empty():
        self.head = new_node
        new_node.next = self.head
    else:
        current = self.head
        while current.next != self.head:
            current = current.next
        current.next = new_node
        new_node.next = self.head
        self.head = new_node
def delete(self, data):
    if self.is_empty():
```

```
            return
        if self.head.data == data:
            current = self.head
            while current.next != self.head:
                current = current.next
            if self.head.next == self.head:
                self.head = None
            else:
                self.head = self.head.next
                current.next = self.head
        else:
            current = self.head
            while current.next != self.head and current.next.data != data:
                current = current.next
            if current.next.data == data:
                current.next = current.next.next
    def display(self):
        elements = []
        current = self.head
        if current:
            repeat = True
            while repeat:
                elements.append(current.data)
                current = current.next
                if current == self.head:
                    repeat = False
```

```python
        print(" -> ".join(map(str, elements)))
    def search(self, target):
        if self.is_empty():
            return False
        current = self.head
        repeat = True
        while repeat:
            if current.data == target:
                return True
            current = current.next
            if current == self.head:
                repeat = False
        return False
# Example usage:
my_circular_linked_list = CircularLinkedList()
my_circular_linked_list.append(1)
my_circular_linked_list.append(2)
my_circular_linked_list.append(3)
my_circular_linked_list.prepend(0)
my_circular_linked_list.display()  # Output: 0 -> 1 -> 2 -> 3
my_circular_linked_list.delete(2)
my_circular_linked_list.display()  # Output: 0 -> 1 -> 3
```

OUTPUT:



Dt:07-06-2024

<u>STACK</u>

Push: adds an element to the top of the stack.

Pop: removes and returns the top element of the stack.

Peek: returns the top element without removing it.

Is_empty: checks if the stack is empty.

Size: returns the number of elements in the stack.

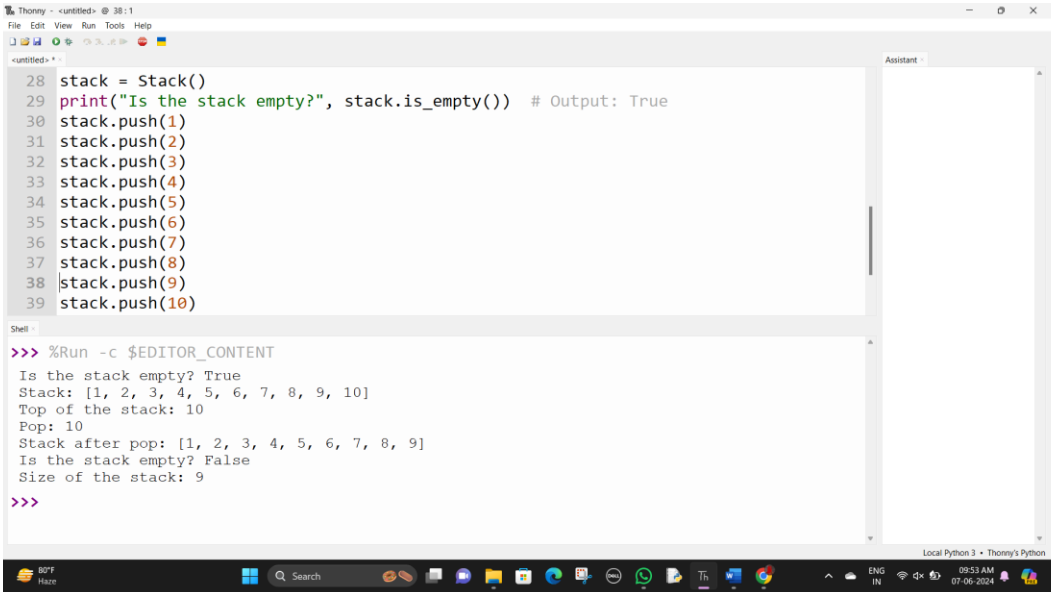
CODE:

```
#STACK

class Stack:

    def __init__(self):

        self.items = []

    def is_empty(self):

        return len(self.items) == 0

    def push(self, item):
```

```
        self.items.append(item)
    def pop(self):
        if not self.is_empty():
            return self.items.pop()
        else:
            raise IndexError("pop from an empty stack")
    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        else:
            raise IndexError("peek from an empty stack")
    def size(self):
        return len(self.items)

# Example usage
stack = Stack()
print("Is the stack empty?", stack.is_empty())  # Output: True
stack.push(1)
stack.push(2)
stack.push(3)
stack.push(4)
stack.push(5)
stack.push(6)
stack.push(7)
stack.push(8)
stack.push(9)
```

```
stack.push(10)

print("Stack:", stack.items)  # Output: [1, 2, 3]

print("Top of the stack:", stack.peek())  # Output: 3

print("Pop:", stack.pop())  # Output: 3 (corrected to call the method)

print("Stack after pop:", stack.items)  # Output: [1, 2]

print("Is the stack empty?", stack.is_empty())  # Output: False

print("Size of the stack:", stack.size())  # Output: 2
```

OUTPUT:



<u>STACK USING LINKED-LIST</u>

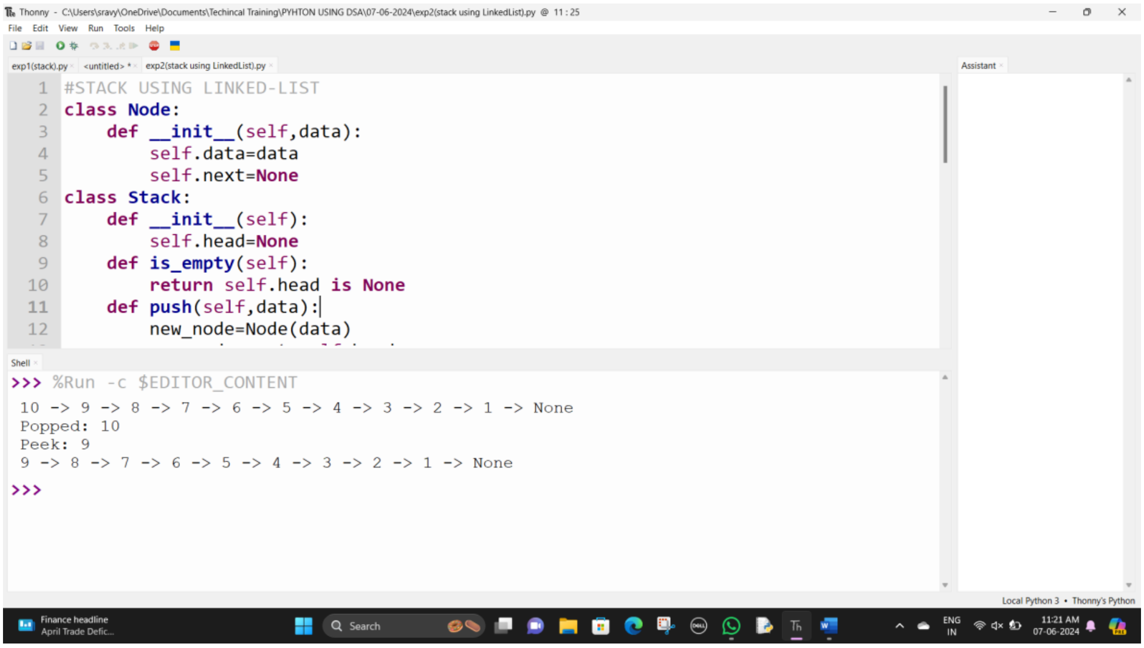
CODE:

```
#STACK USING LINKED-LIST

class Node:

   def __init__(self,data):

       self.data=data

       self.next=None

class Stack:

   def __init__(self):
```

```
        self.head=None
    def is_empty(self):
        return self.head is None
    def push(self,data):
        new_node=Node(data)
        new_node.next=self.head
        self.head=new_node
    def pop(self):
        if self.is_empty():
            return None
        popped=self.head.data
        self.head=self.head.next
        return popped
    def peek(self):
        if self.is_empty():
            return None
        return self.head.data
    def display(self):
        current=self.head
        while current:
            print(current.data, end=" -> ")
            current=current.next
        print("None")
stack=Stack()
stack.push(1)
stack.push(2)
```

```
stack.push(3)

stack.push(4)

stack.push(5)

stack.push(6)

stack.push(7)

stack.push(8)

stack.push(9)

stack.push(10)

stack.display()

print("Popped:", stack.pop())

print("Peek:", stack.peek())

stack.display()
```

OUTPUT:



QUEUE

CODE:

```
#QUEUE
class QueueList:
    def __init__(self):
        self.items = []
    def is_empty(self):
        return len(self.items) == 0
    def enqueue(self, item):
        self.items.append(item)
    def dequeue(self):
        if not self.is_empty():
            return self.items.pop(0)
        else:
            raise IndexError("dequeue from an empty queue")
    def peek(self):
        if not self.is_empty():
            return self.items[0]
        else:
            raise IndexError("peek from an empty queue")
    def size(self):
        return len(self.items)
# Testing the QueueList class
queue = QueueList()
queue.enqueue(1)
queue.enqueue(2)
```

queue.enqueue(3)

queue.enqueue(4)

queue.enqueue(5)

queue.enqueue(6)

queue.enqueue(7)

queue.enqueue(8)

queue.enqueue(9)

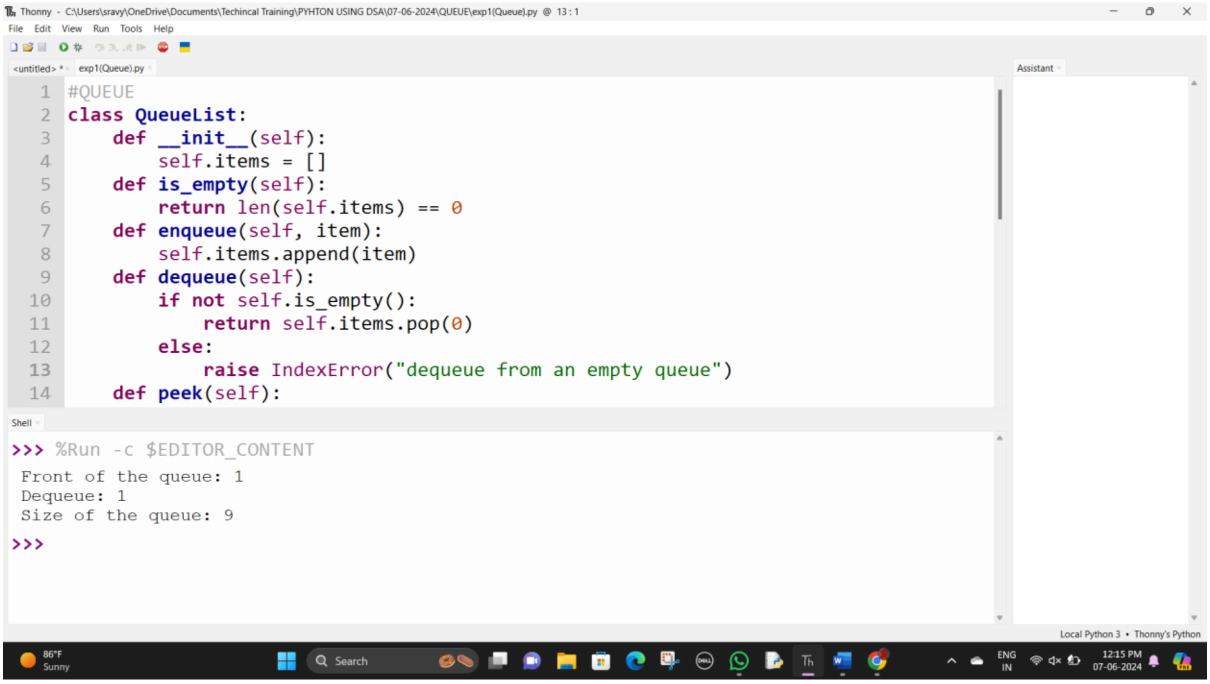
queue.enqueue(10)

print("Front of the queue:", queue.peek())

print("Dequeue:", queue.dequeue())

print("Size of the queue:", queue.size())

OUTPUT:



QUEUE USING LINKED-LIST

CODE:

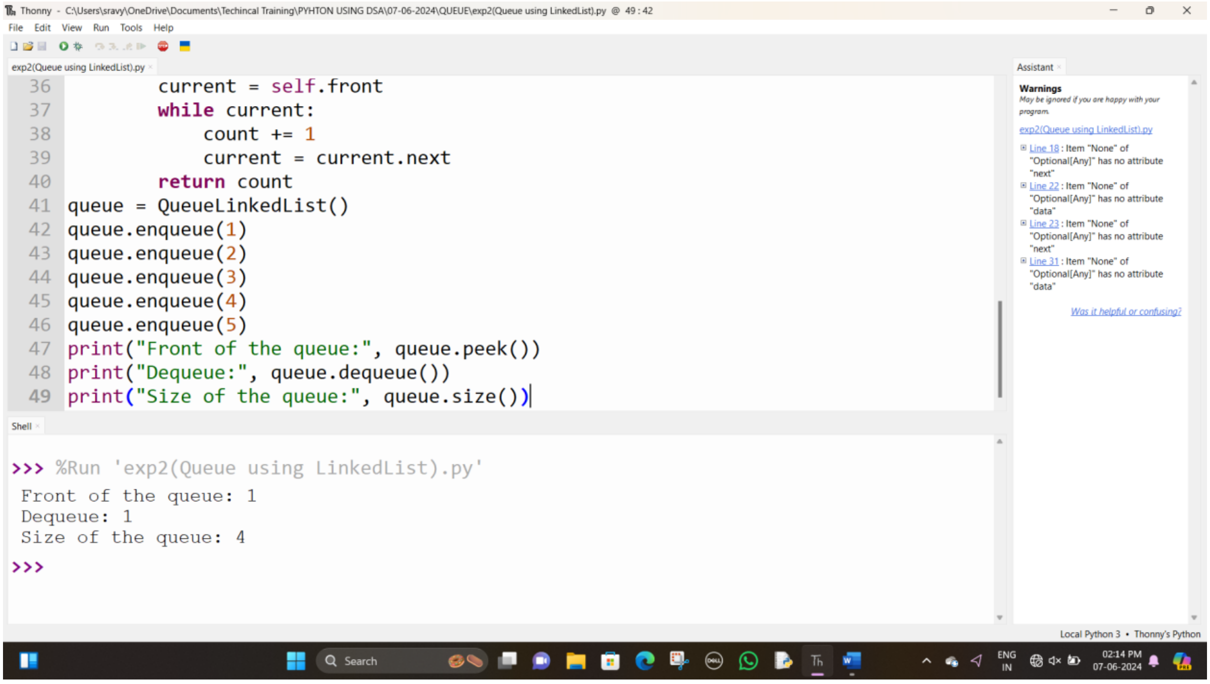
#QUEUE USING LINKED-LIST

class Node:

```
    def __init__(self, data):
        self.data = data
        self.next = None
class QueueLinkedList:
    def __init__(self):
        self.front = None
        self.rear = None
    def is_empty(self):
        return self.front is None
    def enqueue(self, item):
        new_node = Node(item)
        if self.is_empty():
            self.front = new_node
            self.rear = new_node
        else:
            self.rear.next = new_node
            self.rear = new_node
    def dequeue(self):
        if not self.is_empty():
            dequeued_item = self.front.data
            self.front = self.front.next
            if self.front is None:
                self.rear = None
            return dequeued_item
        else:
            raise IndexError("dequeue from an empty queue")
```

```python
    def peek(self):
        if not self.is_empty():
            return self.front.data
        else:
            raise IndexError("peek from an empty queue")
    def size(self):
        count = 0
        current = self.front
        while current:
            count += 1
            current = current.next
        return count
queue = QueueLinkedList()
queue.enqueue(1)
queue.enqueue(2)
queue.enqueue(3)
queue.enqueue(4)
queue.enqueue(5)
print("Front of the queue:", queue.peek())
print("Dequeue:", queue.dequeue())
print("Size of the queue:", queue.size())
```

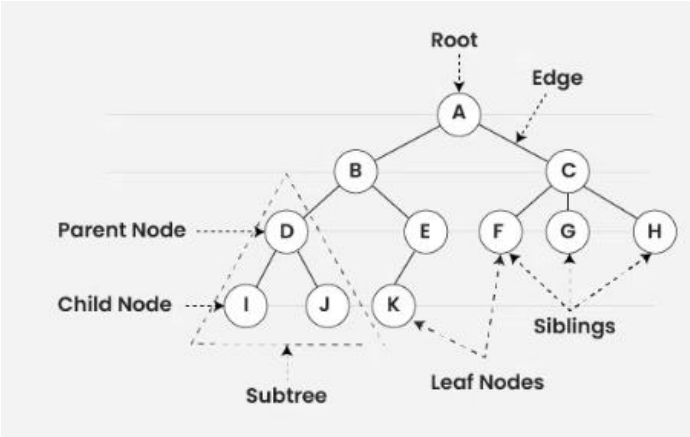
OUTPUT:



Dt:08-06-2024

<u>TREE</u>

- Tree data structure is a hierarchical structure that is used to represent and organize data in a way that is easy to navigate and search. It is a collection of nodes that are connected by edges and has a hierarchical relationship between the nodes.
- The topmost node of the tree is called the root, and the nodes below it are called the child nodes. Each node can have multiple child nodes, and these child nodes can also have their own child nodes, forming a recursive structure.
- The data in a tree are not stored in a sequential manner i.e., they are not stored linearly. Instead, they are arranged on multiple levels or we can say it is a hierarchical structure. For this reason, the tree is considered to be a **non-linear data structure.**



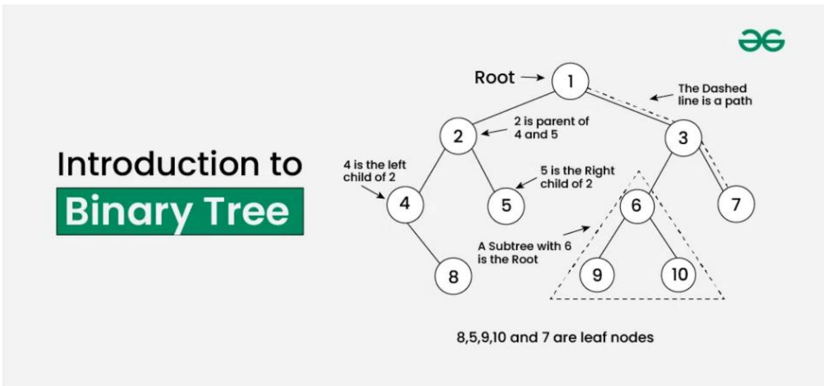
Basic Operations Of Tree Data Structure:

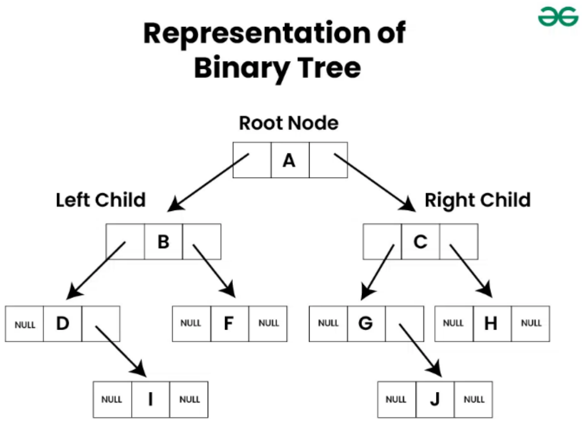
1. **Create** – create a tree in the data structure.
2. **Insert** − Inserts data in a tree.
3. **Search** − Searches specific data in a tree to check whether it is present or not.
4. **Traversal**:
    - Depth-First-Search Traversal
    - Breadth-First-Search Traversal

Types of Tree data:

1. Binary tree:

→Binary Tree is a non-linear data structure where each node has at most two children.





- **Types of Binary Tree:**

➢ Binary Tree consists of following types **based on the number of children**:

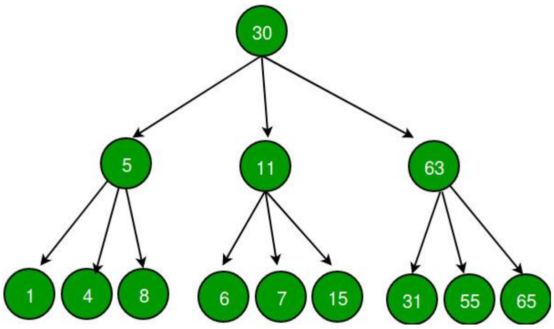
1. Full Binary Tree
2. Degenerate Binary Tree

➢ **On the basis of completion of levels**, the binary tree can be divided into following types:

1. Complete Binary Tree
2. Perfect Binary Tree
3. Balanced Binary Tree

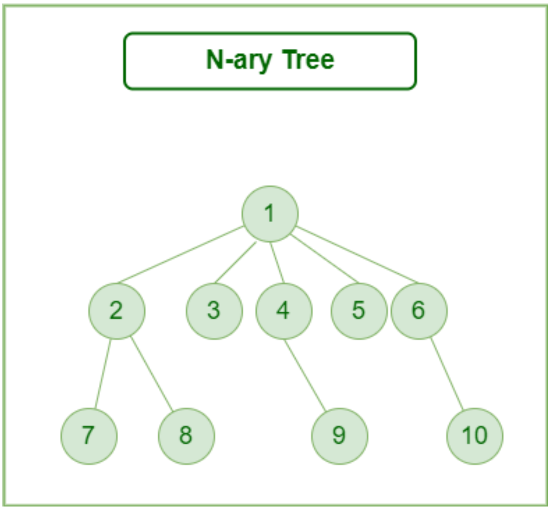
2. Ternary Tree:

→ A Ternary Tree is a tree data structure in which each node has at most three child nodes, usually distinguished as “left”, “mid” and “right”.



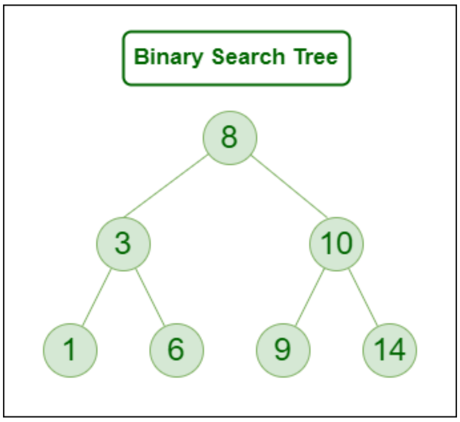
3. Generic / N-aryTree:

→ Generic trees are a collection of nodes where each node is a data structure that consists of records and a list of references to its children(duplicate references are not allowed). Unlike the linked list, each node stores the address of multiple nodes.

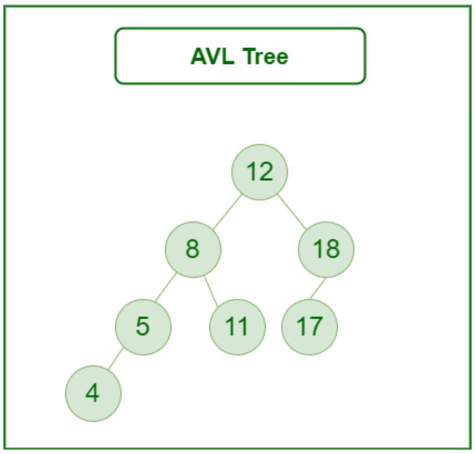


Special Types of Trees:

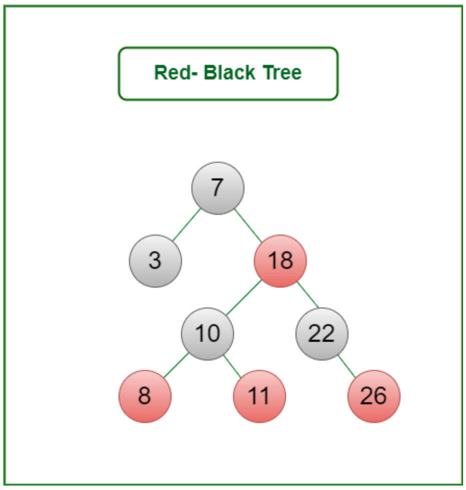
1. **Binary Search Tree.**



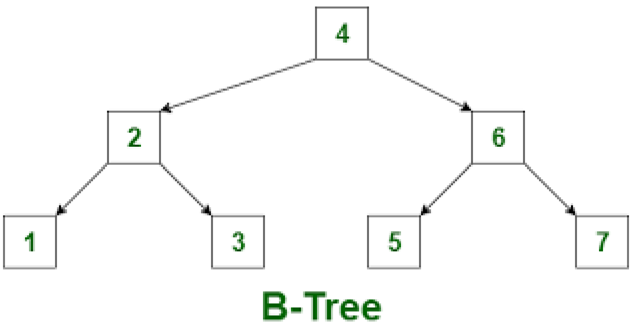
2. **AVL-Tree.**



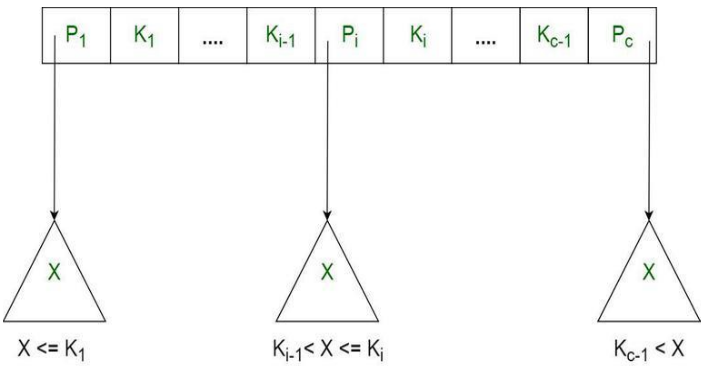
3. **Red-Black Tree.**



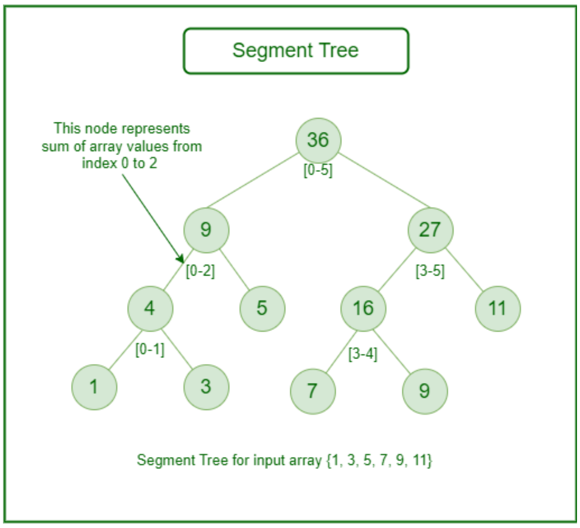
4. **B-Tree.**



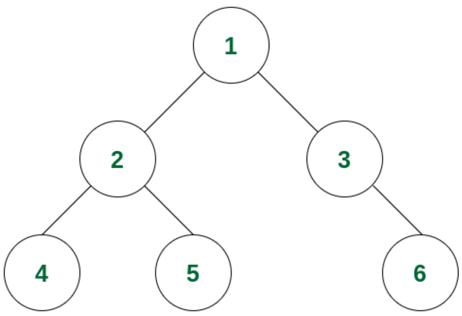
5. **B+ Tree.**



6. Segment Tree.

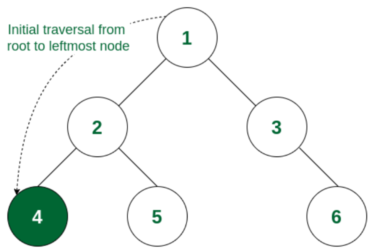


In-order Tree Traversal without Recursion: *(Left-Root-Right)*



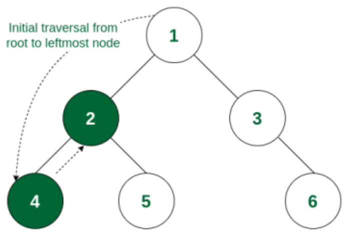
OUTPUT= 4->2->5->1->3->6

Step 1:



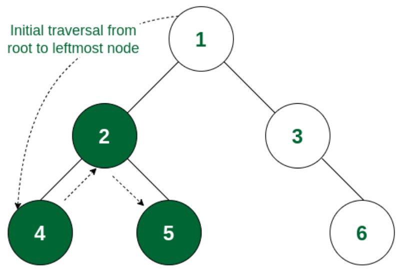
Leftmost node of the tree is visited

Step 2:



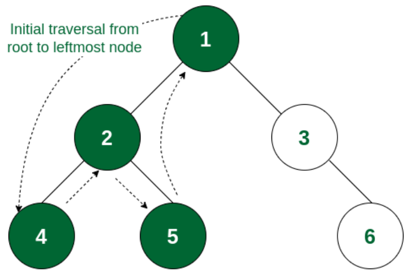
Left subtree of 2 is fully traversed. So 2 is visited next

Step 3:



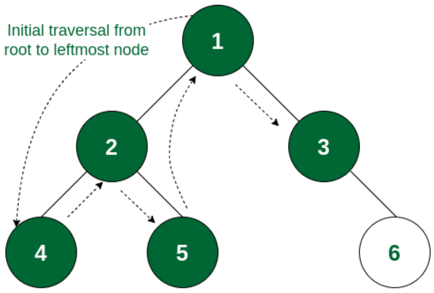
Right subtree of 2 (i.e., 5) is traversed

Step 4:



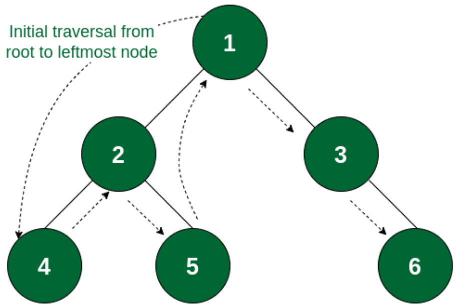
Left subtree of 1 is fully traversed. So 1 is visited next

Step 5:



3 has no left subtree, so it is visited

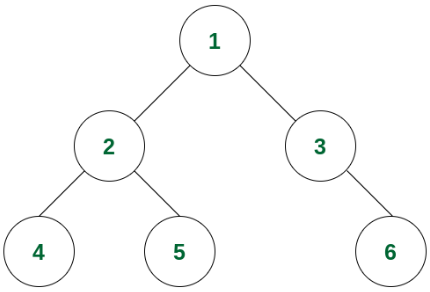
Step 6:



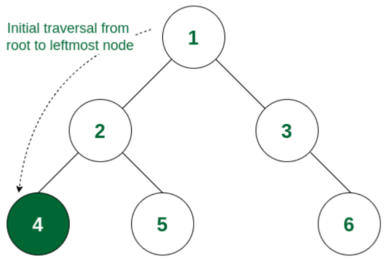
Right Child of 3 is visited

<u>Post-order Traversal of Binary Tree :</u> *(Left-Right-Root)*

EXAMPLE:

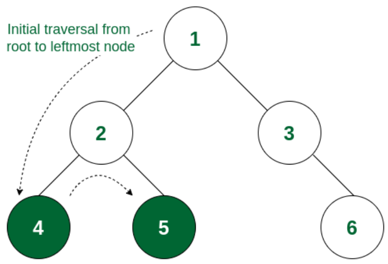


Step 1:



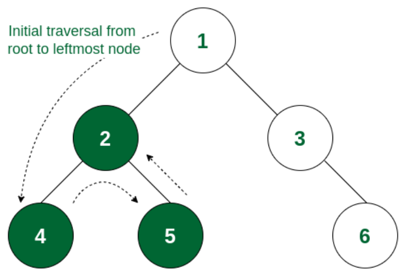
The leftmost leaf node (i.e., 4) is visited first

Step 2:



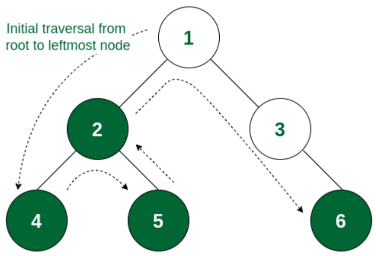
Left subtree of 2 is traversed. So 5 is visited next

Step 3:



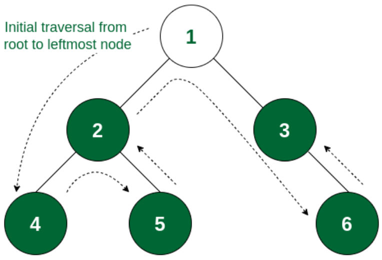
All subtrees of 2 are visited. So 2 is visited next

Step 4:



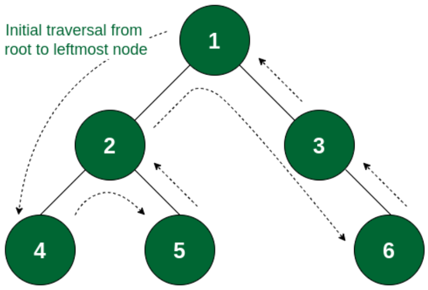
6 has no subtrees. So it is visited

Step 5:



3 is visited after all its subtrees are traversed

Step 6:

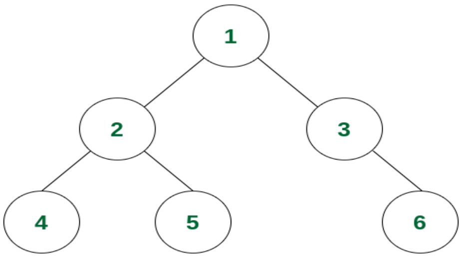


The root of the tree (i.e., 1) is visited

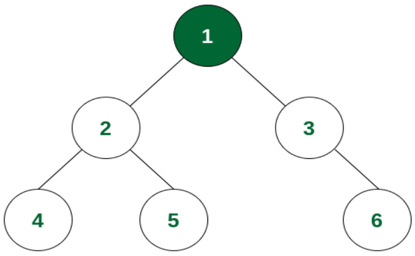
OUTPUT: 4 -> 5 -> 2 -> 6 -> 3 -> 1.

Pre-order Traversal of Binary Tree(*Root-Left-Right)*

EXAMPLE:

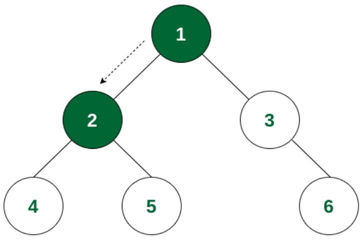


Step 1:



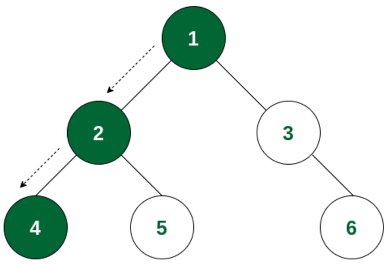
Root of the tree (i.e., 1) is visted

Step 2:



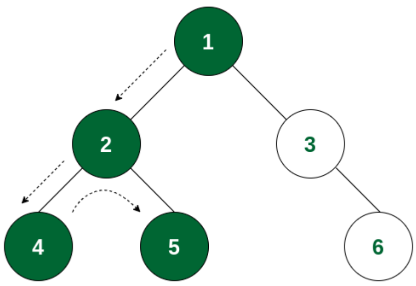
Root of left subtree of 1 (i.e., 2) is visited

Step 3:



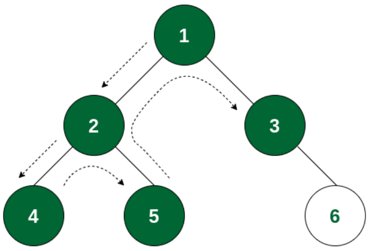
Left child of 2 (i.e., 4) is visited

Step 4:



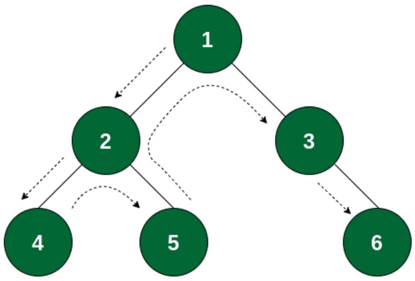
Right child of 2 (i.e., 5) is visited

Step 5:



Root of right subtree of 1 (i.e., 3) is visited

Step 6:



3 has no left subtree. So right subtree is visited

OUTPUT: 1 -> 2 -> 4 -> 5 -> 3 -> 6.

In-order Traversal TREE:

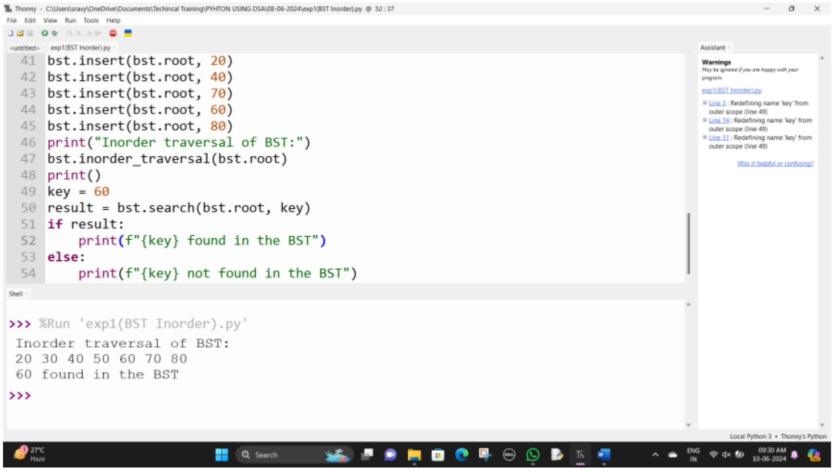
CODE:

```
#Tree

class TreeNode:

    def __init__(self, key):

        self.val = key

        self.left = None

        self.right = None
```

```
#Binary Search Tree
class BST:
    def __init__(self):
        self.root = None

    #Inserting
    def insert(self, root, key):
        if root is None:
            return TreeNode(key)
        else:
            if root.val < key:
                root.right = self.insert(root.right, key)
            else:
                root.left = self.insert(root.left, key)
        return root

    #Inorder Traversal
    def inorder_traversal(self, root):
        if root:
            self.inorder_traversal(root.left)
            print(root.val, end=" ")
            self.inorder_traversal(root.right)

    def search(self, root, key):
        if root is None or root.val == key:
```

```
            return root
        if root.val < key:
            return self.search(root.right, key)
        return self.search(root.left, key)

bst = BST()
bst.root = bst.insert(bst.root, 50)
bst.insert(bst.root, 30)
bst.insert(bst.root, 20)
bst.insert(bst.root, 40)
bst.insert(bst.root, 70)
bst.insert(bst.root, 60)
bst.insert(bst.root, 80)
print("Inorder traversal of BST:")
bst.inorder_traversal(bst.root)
print()
key = 60
result = bst.search(bst.root, key)
if result:
    print(f"{key} found in the BST")
else:
    print(f"{key} not found in the BST")
```

OUTPUT:



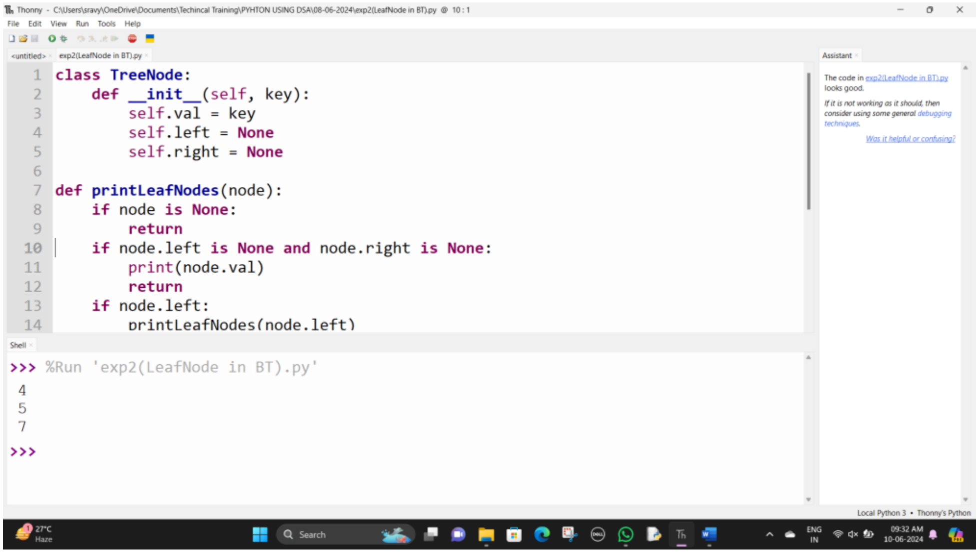
Leaf-Node in Binary Tree:

CODE:

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


def printLeafNodes(node):
    if node is None:
        return
    if node.left is None and node.right is None:
        print(node.val)
        return
    if node.left:
        printLeafNodes(node.left)
    if node.right:
        printLeafNodes(node.right)
```

```
root = TreeNode(1)
root.left = TreeNode(2)
root.right = TreeNode(3)
root.left.left = TreeNode(4)
root.left.right = TreeNode(5)
root.right.right = TreeNode(6)
root.right.right.right = TreeNode(7)
printLeafNodes(root)
```

OUTPUT:



GEEKSFORGEEKS

Given a binary tree, find its height.

Example 1:

Input:

```
   1
  / \
 2   3
```

Output: 2

Example 2:

Input:

```
 2
  \
   1
  /
 3
```

Output: 3

CODE:

```
class Node:
    def init(self,val):
        self.data = val
        self.left = None
        self.right = None
class Solution:
    #Function to find the height of a binary tree.
    def height(self, root):
        if root is None:
            return 0
        return(1+ max(self.height(root.left), self.height(root.right)))
```

OUTPUT:

Compilation Completed

For Input:

1 2 3

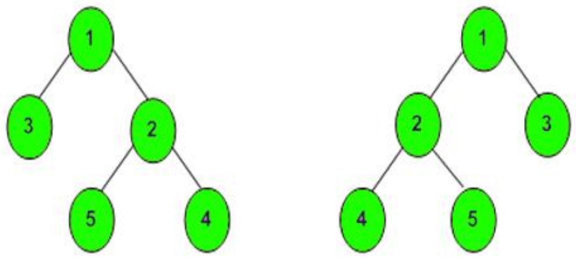
Your Output:

2

Expected Output:

2

Given a Binary Tree, convert it into its mirror.



Mirror Trees

CODE:

```
class Node:
    def init(self, val):
        self.right = None
        self.data = val
        self.left = None
class Solution:
```

```
#Function to convert a binary tree into its mirror tree.
def mirror(self,node):
    if node is None:
        return
    self.mirror(node.left)
    self.mirror(node.right)
    temp=node.left
    node.left=node.right
    node.right=temp
```

OUTPUT:

Compilation Completed

For Input:

1 3 2

Your Output:

2 1 3

Expected Output:

2 1 3

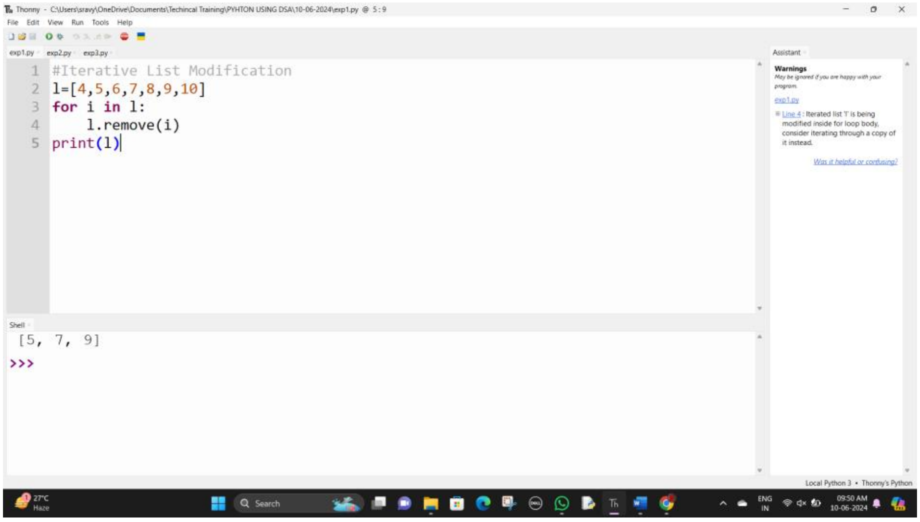
Dt:10-06-2024

1.Iterative List Modification

CODE:

```
l=[4,5,6,7,8,9,10]
for i in l:
    l.remove(i)
print(l)
```

OUTPUT:

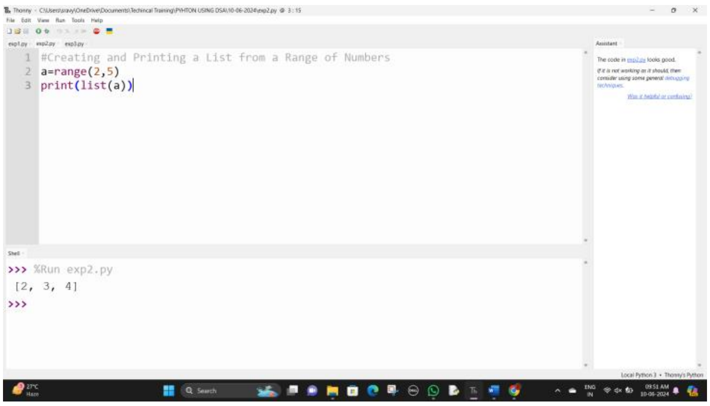


2.Creating and Printing a List from a Range of Numbers

CODE:

```
a=range(2,5)
print(list(a))
```

OUTPUT:



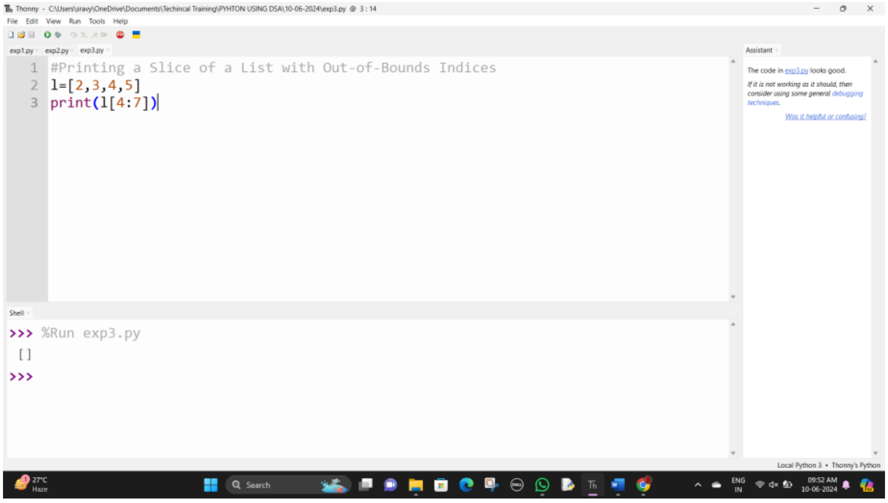
3.Printing a Slice of a List with Out-of-Bounds Indices

CODE:

```
l=[2,3,4,5]
```

print(l[4:7])

OUTPUT:

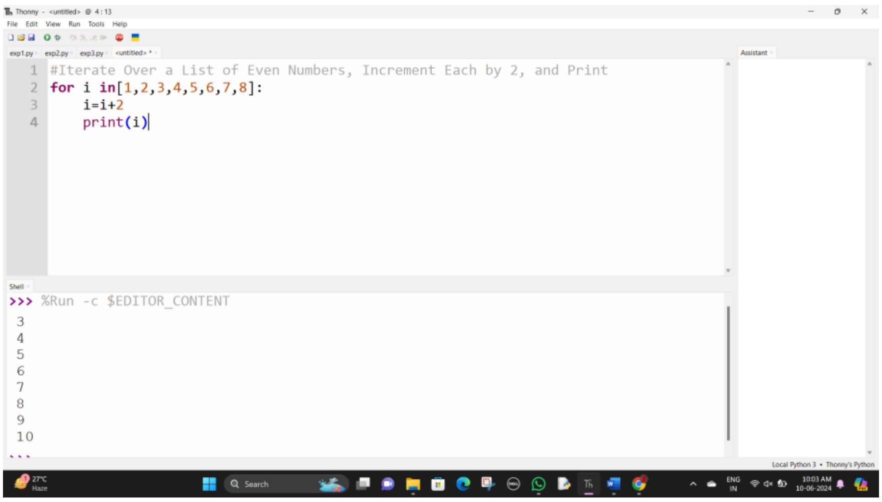


4. Iterate Over a List of Even Numbers, Increment Each by 2, and Print

CODE:

```
for i in[1,2,3,4,5,6,7,8]:
    i=i+2
    print(i)
```

OUTPUT:



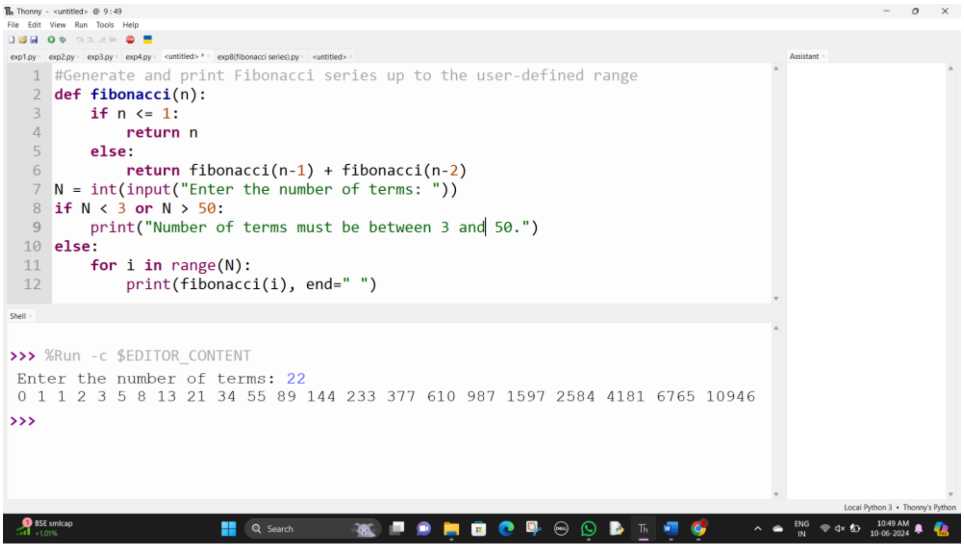
4.Generate and print Fibonacci series up to the user-defined range

CODE:

```
def fibonacci(n):
```

```
    if n <= 1:
        return n
    else:
        return fibonacci(n-1) + fibonacci(n-2)
N = int(input("Enter the number of terms: "))
if N < 3 or N > 50:
    print("Number of terms must be between 3 and 50.")
else:
    for i in range(N):
        print(fibonacci(i), end=" ")
```

OUTPUT:



<u>**LEETCODE**</u>

**1.Given an array of integers nums, return the number of good pairs.
A pair (i, j) is called good if nums[i] == nums[j] and i < j.**

CODE:

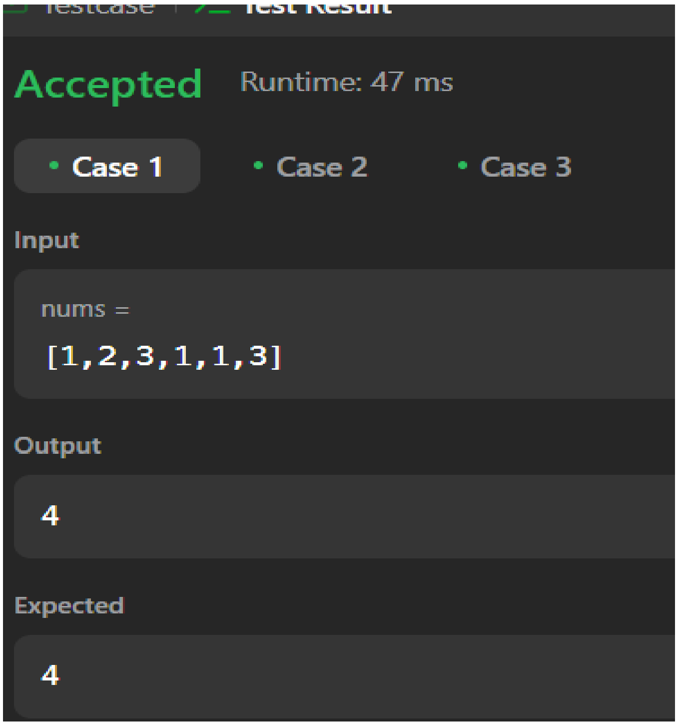
```
class Solution:
    def numIdenticalPairs(self, nums: List[int]) -> int:
        ans = 0
        for i in range(len(nums)-1):# i is indexing from 0 to n-1
```

```
            for j in range(i+1,len(nums)):# j is from i+1 to n

                if nums[i] == nums[j]:# i < j always

                    ans +=1

        return ans
```

OR

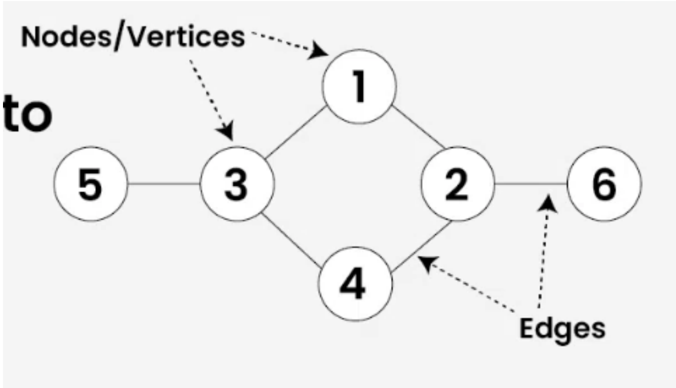
```
class Solution:
    def numIdenticalPairs(self, nums: List[int]) -> int:
      hashMap={}
      res = 0
      for number in nums:
        if number in hashMap:
          res+=hashMap[number]
          hashMap[number]+=1
        else:
          hashMap[number]=1
      return res
```

OUTPUT:



GRAPH

- **Graph Data Structure** is a non-linear data structure consisting of vertices and edges. It is useful in fields such as social network analysis, recommendation systems, and computer networks. In the field of sports data science, graph data structure can be used to analyze and understand the dynamics of team performance and player interactions on the field.



COMPONENTS OF GRAPH

Vertices: Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labeled or unlabelled.

Edges: Edges are drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph. Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be labelled/unlabelled.

REPRESENTATION OF GRAPHS

Here are the two most common ways to represent a graph :

1.Adjacency Matrix

2.Adjacency List

1.Adjacency Matrix: An adjacency matrix is a way of representing a graph as a matrix of boolean (0's and 1's).

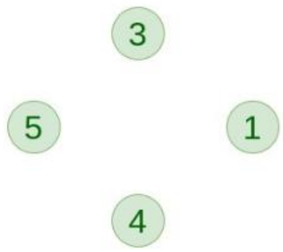
- If there is an edge from vertex i to j, mark adjMat[i][j] as 1.
- If there is no edge from vertex i to j, mark adjMat[i][j] as 0.

2.Adjacency List: An array of Lists is used to store edges between two vertices. The size of array is equal to the number of vertices (i.e, n). Each index in this array represents a specific vertex in the graph. The entry at the index i of the array contains a linked list containing the vertices that are adjacent to vertex i.

- adjList[0] will have all the nodes which are connected (neighbour) to vertex 0.
- adjList[1] will have all the nodes which are connected (neighbour) to vertex 1 and so on.

TYPES OF GRAPHS

1.Null Graph: A graph is known as a null graph if there are no edges in the graph.

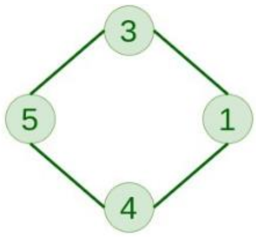


Null Graph

2. Trivial Graph: Graph having only a single vertex, it is also the smallest graph possible.

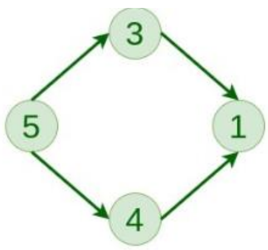
Trivial Graph

3. Undirected Graph: A graph in which edges do not have any direction. That is the nodes are unordered pairs in the definition of every edge.



Undirected Graph

4. Directed Graph: A graph in which edge has direction. That is the nodes are ordered pairs in the definition of every edge.



Directed Graph

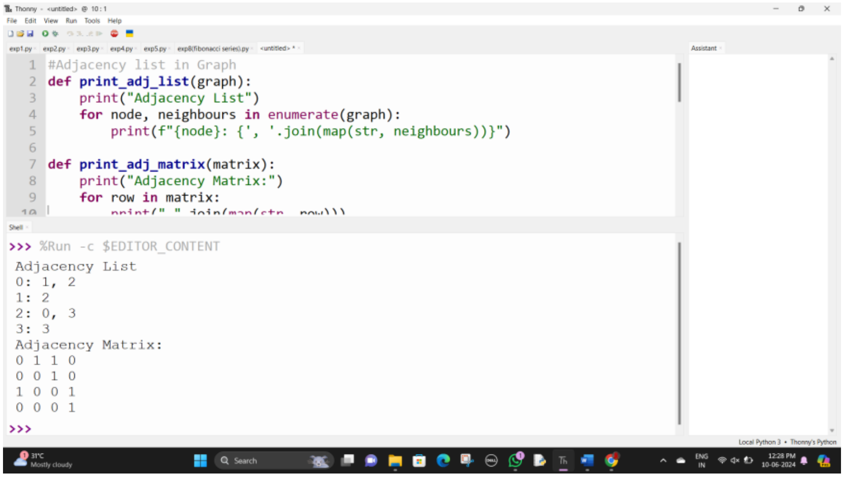
ADJACENCY LIST

CODE:

```
def print_adj_list(graph):
    print("Adjacency List")
    for node, neighbours in enumerate(graph):
        print(f"{node}: {', '.join(map(str, neighbours))}")
def print_adj_matrix(matrix):
    print("Adjacency Matrix:")
    for row in matrix:
        print(" ".join(map(str, row)))
adj_list = [
    [1, 2],  # neighbours of node 0
```

```
    [2],     # neighbours of node 1
    [0, 3],  # neighbours of node 2
    [3]      # neighbours of node 3
]
edges = [
    (0, 1),
    (0, 2),
    (1, 2),
    (2, 0),
    (2, 3),
    (3, 3)
]
num_nodes = len(adj_list)
adj_matrix = [[0] * num_nodes for _ in range(num_nodes)]
for src, dest in edges:
    adj_matrix[src][dest] = 1
print_adj_list(adj_list)
print_adj_matrix(adj_matrix)
```

OUTPUT:



LEETCODE

1.There are n kids with candies. You are given an integer array candies, where each candies[i] represents the number of candies the ith kid has, and an integer extraCandies, denoting the number of extra candies that you have. Return a boolean array result of length n, where result[i] is true if, after giving the ith kid all the extraCandies, they will have the greatest number of candies among all the kids, or false otherwise.
Note that multiple kids can have the greatest number of candies.
CODE:

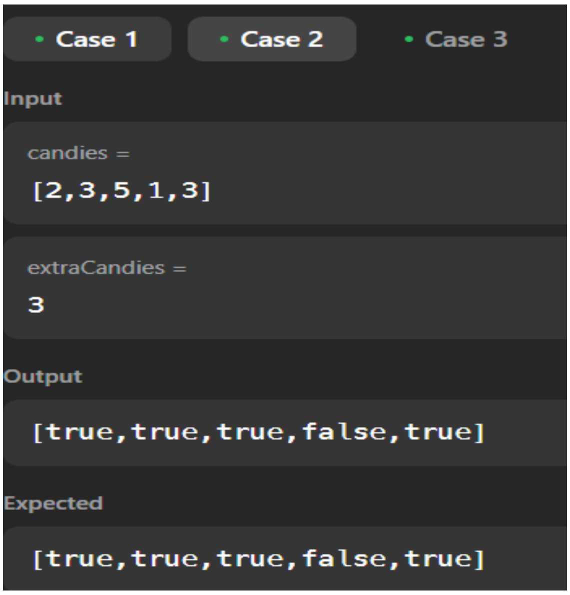
```
class Solution:
  def kidsWithCandies(self, candies: List[int], extraCandies: int) -> List[bool]:
    max_candies = max(candies)  # Find the maximum candies any kid has
    result = []  # Initialize the result list
        for candy in candies:
      # Check if giving the current kid all the extra candies makes them have
the most candies
      if candy + extraCandies >= max_candies:
        result.append(True)  # If true, append True to result
      else:
        result.append(False)  # Otherwise, append False to result
    return result  # Return the result list
```

OR

```
class Solution:
  def kidsWithCandies(self, candies: List[int], extraCandies: int) -> List[bool]:
   result_lst=[]
   max_num=max(candies)
```

```
  for num in candies:
    result_lst.append(num+extraCandies>=max_num)
  return result_lst
```

OUTPUT:



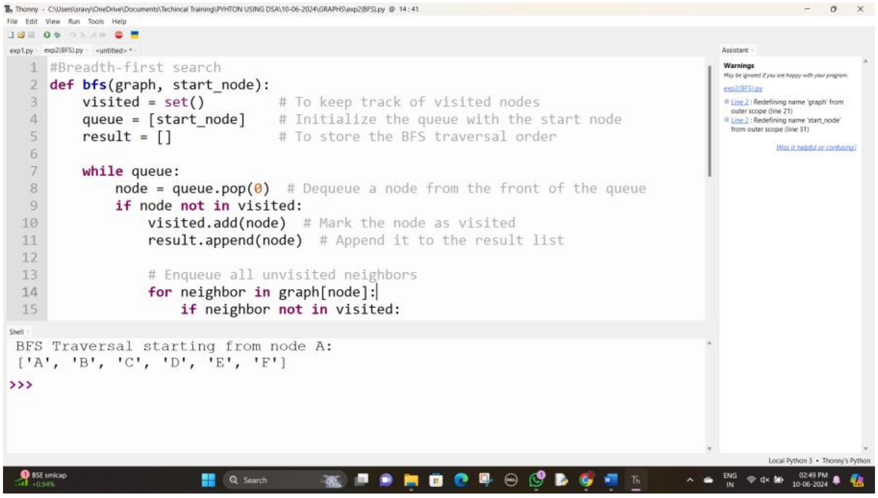
BREADTH-FIRST SEARCH

CODE:

```
#Breadth-first search

def bfs(graph, start_node):

  visited = set()       # To keep track of visited nodes

  queue = [start_node]    # Initialize the queue with the start node

  result = []         # To store the BFS traversal order

  while queue:

    node = queue.pop(0)  # Dequeue a node from the front of the queue
```

```
        if node not in visited:
            visited.add(node)  # Mark the node as visited
            result.append(node)  # Append it to the result list
            # Enqueue all unvisited neighbors
            for neighbor in graph[node]:
                if neighbor not in visited:
                    queue.append(neighbor)
    return result
# Define the graph using an adjacency list representation
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}
# Define the start node for BFS traversal
start_node = 'A'
# Print the result of BFS traversal
print("BFS Traversal starting from node A:")
print(bfs(graph, start_node))
```

OUTPUT:



DEPTH-FIRST SEARCH

CODE:

```
#Depth-first search
def dfs(graph, start_node):
    visited = set()        # To keep track of visited nodes
    stack = [start_node]    # Initialize the stack with the start node
    result = []            # To store the BFS traversal order
    while stack:
        node = stack.pop()  # Destack a node from the front of the stack
        if node not in visited:
            visited.add(node)  # Mark the node as visited
            result.append(node)  # Append it to the result list
            # Enstack all unvisited neighbors
            for neighbor in graph[node]:
                if neighbor not in visited:
                    stack.append(neighbor)


    return result
```

```
# Define the graph using an adjacency list representation
graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D', 'E'],
    'C': ['A', 'F'],
    'D': ['B'],
    'E': ['B', 'F'],
    'F': ['C', 'E']
}
# Define the start node for BFS traversal
start_node = 'A'
# Print the result of BFS traversal
print("DFS traversal order:", dfs(graph, start_node))
```

OUTPUT: