ll insert at beginning

In [1]:

```
class Node:
    def __init__(self,data):
        self.data=data
        self.next=None
class LL:
    def __init__(self):
        self.head=None
    def insrert_at_begining(self,data):
        new_node=Node(data)
        new_node.next = self.head
        self.head = new_node

    def display(self):
        current = self.head
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")
L_L=LL()
L_L.insrert_at_begining(1)
L_L.insrert_at_begining(2)
L_L.insrert_at_begining(3)
L_L.insrert_at_begining(4)
L_L.insrert_at_begining(5)

L_L.display()
```

```
5 -> 4 -> 3 -> 2 -> 1 -> None
```

insert at end

In [2]:
```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
class LinkedList:
    def __init__(self):
        self.head = None
    def insert_at_end(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
            return
        current = self.head
        while current.next:
            current = current.next
        current.next = new_node
    def display(self):
        current = self.head
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")
linked_list = LinkedList()
linked_list.insert_at_end(1)
linked_list.insert_at_end(2)
linked_list.insert_at_end(3)
linked_list.display()
```

```
1 -> 2 -> 3 -> None
```

insert at position

In [3]:
```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

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

    def insert_at_position(self, data, position):
        new_node = Node(data)
        if position == 1:
            new_node.next = self.head
            self.head = new_node
        else:
            current = self.head
            count = 1
            while count < position - 1 and current is not None:
                current = current.next
                count += 1
            if current is None:
                print("Position is out of range.")
                return
            new_node.next = current.next
            current.next = new_node

    def display(self):
        current = self.head
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")

if __name__ == "__main__":
    linked_list = LinkedList()

    linked_list.insert_at_position(1, 1)
    linked_list.insert_at_position(2, 2)
    linked_list.insert_at_position(3, 1)
    linked_list.insert_at_position(4, 4)

    print("Linked List: ")
    linked_list.display()
```

```
Linked List: 
3 -> 1 -> 2 -> 4 -> None
```

doubly ll

In [4]:
```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None

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

    def prepend(self, data):
        new_node = Node(data)
        if not self.head:
            self.head = new_node
        else:
            new_node.next = self.head
            self.head.prev = new_node
            self.head = new_node

    def delete(self, data):
        current = self.head
        while current:
            if current.data == data:
                if current.prev:
                    current.prev.next = current.next
                else:
                    self.head = current.next
                if current.next:
                    current.next.prev = current.prev
                return
            current = current.next

    def display(self):
        current = self.head
        while current:
            print(current.data, end=" <-> ")
            current = current.next
        print("None")

# Example usage:
dllist = DoublyLinkedList()
dllist.prepend(1)
dllist.prepend(2)
dllist.prepend(3)
dllist.prepend(4)
dllist.prepend(0)
dllist.display()

dllist.delete(2)
dllist.display()
```

```
0 <-> 4 <-> 3 <-> 2 <-> 1 <-> None
0 <-> 4 <-> 3 <-> 1 <-> None
```

delete at beginning in sll

In [5]:
```
# class Node:
def __init__(self, data):
        self.data = data
        self.next = None

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

    def insert_at_beginning(self,data):#insert_at_beginning
        new_node=Node(data)
        new_node.next = self.head
        self.head = new_node

    def delete_at_beginning(self):
        if not self.head:  # Check if the linked list is empty
            print("Linked list is empty. Nothing to delete.")
            return

        # Update the head to point to the second node (new head)
        self.head = self.head.next

    def display(self):
        current = self.head
        while current:
            print(current.data, end=" -> ")
            current = current.next
        print("None")

# Create a linked list
linked_list = LinkedList()

# Insert elements at the beginning
linked_list.insert_at_beginning(3)
linked_list.insert_at_beginning(2)
linked_list.insert_at_beginning(1)

# Display the linked list
print("Linked List before deletion:")
linked_list.display()

# Delete the node at the beginning
linked_list.delete_at_beginning()

# Display the linked list after deletion
print("Linked List after deletion:")
linked_list.display()
```

```
Linked List before deletion:
1 -> 2 -> 3 -> None
Linked List after deletion:
2 -> 3 -> None
```

stack dynamically

In [6]:
```
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:
            print("Stack is empty. Cannot pop.")

    def peek(self):
        if not self.is_empty():
            return self.items[-1]
        else:
            print("Stack is empty. Cannot peek.")

    def size(self):
        return len(self.items)

if __name__ == "__main__":
    stack = Stack()

    while True:
        print("\nStack Operations:")
        print("1. Push")
        print("2. Pop")
        print("3. Peek")
        print("4. Check if empty")
        print("5. Exit")

        choice = input("Enter your choice (1/2/3/4/5): ")

        if choice == '1':
            item = input("Enter the item to push: ")
            stack.push(item)
            print(f"{item} pushed onto the stack.")
        elif choice == '2':
            popped_item = stack.pop()
            if popped_item is not None:
                print(f"Popped item: {popped_item}")
        elif choice == '3':
            top_item = stack.peek()
            if top_item is not None:
                print(f"Top item: {top_item}")
        elif choice == '4':
            if stack.is_empty():
                print("Stack is empty.")
            else:
                print("Stack is not empty.")
        elif choice == '5':
            print("Exiting the program.")
            break
        else:
            print("Invalid choice. Please enter a valid option.")
```

```
Stack Operations:
1. Push
2. Pop
3. Peek
4. Check if empty
5. Exit
Enter your choice (1/2/3/4/5): 5
Exiting the program.
```

queue statically

In [8]:
```
class StaticQueue:
    def __init__(self, capacity):
        self.capacity = capacity
        self.queue = [None] * capacity
        self.front = 0
        self.rear = -1
        self.size = 0

    def is_empty(self):
        return self.size == 0

    def is_full(self):
        return self.size == self.capacity

    def enqueue(self, item):
        if self.is_full():
            print("Queue is full. Cannot enqueue.")
        else:
            self.rear = (self.rear + 1) % self.capacity
            self.queue[self.rear] = item
            self.size += 1

    def dequeue(self):
        if self.is_empty():
            print("Queue is empty. Cannot dequeue.")
            return None
        else:
            item = self.queue[self.front]
            self.queue[self.front] = None
            self.front = (self.front + 1) % self.capacity
            self.size -= 1
            return item

    def peek(self):
        if self.is_empty():
            print("Queue is empty. Cannot peek.")
            return None
        else:
            return self.queue[self.front]

if __name__ == "__main__":
    queue = StaticQueue(5)

    queue.enqueue(1)
    queue.enqueue(2)
    queue.enqueue(3)
    queue.enqueue(4)

    print("Queue: ", queue.queue)

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

    print("Peek:", queue.peek())
    print("Queue size:", queue.size)

    print("Is the queue empty?", queue.is_empty())
    print("Is the queue full?", queue.is_full())
```

```
Queue:  [1, 2, 3, 4, None]
Dequeue: 1
Queue after dequeue:  [None, 2, 3, 4, None]
Peek: 2
Queue size: 3
Is the queue empty? False
Is the queue full? False
```

dynamic queue

In [7]:

```python
class Queue:
    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:
            print("Queue is empty. Cannot dequeue.")

    def peek(self):
        if not self.is_empty():
            return self.items[0]
        else:
            print("Queue is empty. Cannot peek.")

    def size(self):
        return len(self.items)

if __name__ == "__main__":
    queue = Queue()

    while True:
        print("\nQueue Operations:")
        print("1. Enqueue")
        print("2. Dequeue")
        print("3. Peek")
        print("4. Check if empty")
        print("5. Exit")

        choice = input("Enter your choice (1/2/3/4/5): ")

        if choice == '1':
            item = input("Enter the item to enqueue: ")
            queue.enqueue(item)
            print(f"{item} enqueued into the queue.")
        elif choice == '2':
            dequeued_item = queue.dequeue()
            if dequeued_item is not None:
                print(f"Dequeued item: {dequeued_item}")
        elif choice == '3':
            front_item = queue.peek()
            if front_item is not None:
                print(f"Front item: {front_item}")
        elif choice == '4':
            if queue.is_empty():
                print("Queue is empty.")
            else:
                print("Queue is not empty.")
        elif choice == '5':
            print("Exiting the program.")
            break
        else:
            print("Invalid choice. Please enter a valid option.")
```

```
Queue Operations:
1. Enqueue
2. Dequeue
3. Peek
4. Check if empty
5. Exit
Enter your choice (1/2/3/4/5): 5
Exiting the program.
```

static stack

In [9]:
```
class StaticStack:
    def __init__(self, capacity):
        self.capacity = capacity
        self.stack = [None] * capacity
        self.top = -1

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

    def is_full(self):
        return self.top == self.capacity - 1

    def push(self, item):
        if self.is_full():
            print("Stack is full. Cannot push.")
        else:
            self.top += 1
            self.stack[self.top] = item

    def pop(self):
        if self.is_empty():
            print("Stack is empty. Cannot pop.")
            return None
        else:
            item = self.stack[self.top]
            self.stack[self.top] = None
            self.top -= 1
            return item

    def peek(self):
        if self.is_empty():
            print("Stack is empty. Cannot peek.")
            return None
        else:
            return self.stack[self.top]

if __name__ == "__main__":
    stack = StaticStack(5)

    stack.push(1)
    stack.push(2)
    stack.push(3)
    stack.push(4)

    print("Stack: ", stack.stack)

    print("Pop:", stack.pop())
    print("Stack after pop: ", stack.stack)

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

    print("Is the stack empty?", stack.is_empty())
    print("Is the stack full?", stack.is_full())
```

```
Stack:  [1, 2, 3, 4, None]
Pop: 4
Stack after pop:  [1, 2, 3, None, None]
Peek: 3
Is the stack empty? False
Is the stack full? False
```

Write a Python program for conversion of Infix expression to Postfix

In [16]:

```python
def infix_to_postfix(infix_expression):
    precedence = {'+': 1, '-': 1, '*': 2, '/': 2, '^': 3}
    output = []
    stack = []

    def has_higher_precedence(op1, op2):
        return precedence[op1] >= precedence[op2]

    for token in infix_expression.split():
        if token.isalnum():
            output.append(token)
        elif token == '(':
            stack.append(token)
        elif token == ')':
            while stack and stack[-1] != '(':
                output.append(stack.pop())
            stack.pop()  # Pop the '('
        else:
            while stack and stack[-1] != '(' and has_higher_precedence(stack
                output.append(stack.pop())
            stack.append(token)

    while stack:
        output.append(stack.pop())

    return ' '.join(output)

if __name__ == "__main__":
    infix_expression = input("Enter an infix expression: ")
    postfix_expression = infix_to_postfix(infix_expression)
    print("Postfix expression:", postfix_expression)
```

```
Enter an infix expression: 2+5-2*6/3
Postfix expression: 2+5-2*6/3
```

Implement Binary Search Tree (BST) to perform following operations on BST–Create, Recursive Traversals -Inorder, Preorder, Postorder

In [20]:
```
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

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

    def insert(self, key):
        self.root = self._insert(self.root, key)

    def _insert(self, root, key):
        if root is None:
            return Node(key)
        if key < root.key:
            root.left = self._insert(root.left, key)
        elif key > root.key:
            root.right = self._insert(root.right, key)
        return root

    def inorder(self):
        self._inorder(self.root)
        print()

    def _inorder(self, root):
        if root:
            self._inorder(root.left)
            print(root.key, end=' ')
            self._inorder(root.right)

    def preorder(self):
        self._preorder(self.root)
        print()

    def _preorder(self, root):
        if root:
            print(root.key, end=' ')
            self._preorder(root.left)
            self._preorder(root.right)

    def postorder(self):
        self._postorder(self.root)
        print()

    def _postorder(self, root):
        if root:
            self._postorder(root.left)
            self._postorder(root.right)
            print(root.key, end=' ')

if __name__ == "__main__":
    bst = BinarySearchTree()
    keys = [50, 30, 70, 20, 40, 60, 80]

    for key in keys:
        bst.insert(key)

    print("Inorder Traversal:")
    bst.inorder()
```

```
    print("Preorder Traversal:")
    bst.preorder()

    print("Postorder Traversal:")
    bst.postorder()
 #  In this program:

#   We have a Node class to represent the nodes of the binary search tree.
#   The BinarySearchTree class has methods to perform the following operati
#   insert: Inserts a key into the BST.
#   _insert: A helper function for inserting a key.
#   inorder: Performs an inorder traversal of the BST.
#   preorder: Performs a preorder traversal of the BST.
#   postorder: Performs a postorder traversal of the BST.
#   In the if __name__ == "__main__": block, we create a BST, insert a list
#   and then perform the three recursive traversals (inorder, preorder, and
#   elements in the tree in different orders.
```

```
Inorder Traversal:
20 30 40 50 60 70 80
Preorder Traversal:
50 30 20 40 70 60 80
Postorder Traversal:
20 40 30 60 80 70 50
```

Implement a BST to perform following operations : insert, delete and create mirror image of BST

In [22]:
```
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

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

    def insert(self, key):
        self.root = self._insert(self.root, key)

    def _insert(self, root, key):
        if root is None:
            return Node(key)
        if key < root.key:
            root.left = self._insert(root.left, key)
        elif key > root.key:
            root.right = self._insert(root.right, key)
        return root

    def delete(self, key):
        self.root = self._delete(self.root, key)

    def _delete(self, root, key):
        if root is None:
            return root

        if key < root.key:
            root.left = self._delete(root.left, key)
        elif key > root.key:
            root.right = self._delete(root.right, key)
        else:
            if root.left is None:
                return root.right
            elif root.right is None:
                return root.left

            root.key = self._get_min_value(root.right)
            root.right = self._delete(root.right, root.key)

        return root

    def create_mirror(self):
        self.root = self._create_mirror(self.root)

    def _create_mirror(self, root):
        if root is None:
            return root

        root.left, root.right = root.right, root.left
        self._create_mirror(root.left)
        self._create_mirror(root.right)

        return root

    def _get_min_value(self, node):
        while node.left is not None:
            node = node.left
        return node.key
```

```
    def inorder(self):
        self._inorder(self.root)
        print()

    def _inorder(self, root):
        if root:
            self._inorder(root.left)
            print(root.key, end=' ')
            self._inorder(root.right)

if __name__ == "__main__":
    bst = BinarySearchTree()
    keys = [50, 30, 70, 20, 40, 60, 80]

    for key in keys:
        bst.insert(key)

    print("Inorder Traversal:")
    bst.inorder()

    bst.delete(30)
    print("Inorder Traversal after deleting 30:")
    bst.inorder()

    bst.create_mirror()
    print("Inorder Traversal of the mirror image:")
    bst.inorder()
```

```
Inorder Traversal:
20 30 40 50 60 70 80
Inorder Traversal after deleting 30:
20 40 50 60 70 80
Inorder Traversal of the mirror image:
80 70 60 50 40 20
```

Implement BST for counting leaf, non-leaf and total nodes.

In [23]:
```
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

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

    def insert(self, key):
        self.root = self._insert(self.root, key)

    def _insert(self, root, key):
        if root is None:
            return Node(key)
        if key < root.key:
            root.left = self._insert(root.left, key)
        elif key > root.key:
            root.right = self._insert(root.right, key)
        return root

    def count_leaf_nodes(self):
        return self._count_leaf_nodes(self.root)

    def _count_leaf_nodes(self, node):
        if node is None:
            return 0
        if node.left is None and node.right is None:
            return 1
        return self._count_leaf_nodes(node.left) + self._count_leaf_nodes(nc

    def count_non_leaf_nodes(self):
        return self._count_non_leaf_nodes(self.root)

    def _count_non_leaf_nodes(self, node):
        if node is None:
            return 0
        if node.left is not None or node.right is not None:
            return (
                1
                + self._count_non_leaf_nodes(node.left)
                + self._count_non_leaf_nodes(node.right)
            )
        return 0

    def count_total_nodes(self):
        return self._count_total_nodes(self.root)

    def _count_total_nodes(self, node):
        if node is None:
            return 0
        return 1 + self._count_total_nodes(node.left) + self._count_total_nc

if __name__ == "__main__":
    bst = BinarySearchTree()
    keys = [50, 30, 70, 20, 40, 60, 80]

    for key in keys:
        bst.insert(key)
```

```
    print("Total Nodes:", bst.count_total_nodes())
    print("Leaf Nodes:", bst.count_leaf_nodes())
    print("Non-Leaf Nodes:", bst.count_non_leaf_nodes())
```

```
Total Nodes: 7
Leaf Nodes: 4
Non-Leaf Nodes: 3
```

Write a Python program for sorting integer array using: Bubble Sort, Selection sort, Insertion Sort, Quick Sort, Merge sort

In [24]:
```
# Bubble Sort
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]

# Selection Sort
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]

# Insertion Sort
def insertion_sort(arr):
    n = len(arr)
    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

# Quick Sort
def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[len(arr) // 2]
    left = [x for x in arr if x < pivot]
    middle = [x for x in arr if x == pivot]
    right = [x for x in arr if x > pivot]
    return quick_sort(left) + middle + quick_sort(right)

# Merge Sort
def merge_sort(arr):
    if len(arr) <= 1:
        return arr
    mid = len(arr) // 2
    left = arr[:mid]
    right = arr[mid:]
    left = merge_sort(left)
    right = merge_sort(right)
    return list(merge(left, right))

def merge(left, right):
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result += left[i:]
```

```
    result += right[j:]
    return result

if __name__ == "__main__":
    arr = [64, 25, 12, 22, 11]

    # Bubble Sort
    bubble_sorted = arr.copy()
    bubble_sort(bubble_sorted)
    print("Bubble Sort:", bubble_sorted)

    # Selection Sort
    selection_sorted = arr.copy()
    selection_sort(selection_sorted)
    print("Selection Sort:", selection_sorted)

    # Insertion Sort
    insertion_sorted = arr.copy()
    insertion_sort(insertion_sorted)
    print("Insertion Sort:", insertion_sorted)

    # Quick Sort
    quick_sorted = arr.copy()
    quick_sorted = quick_sort(quick_sorted)
    print("Quick Sort:", quick_sorted)

    # Merge Sort
    merge_sorted = arr.copy()
    merge_sorted = merge_sort(merge_sorted)
    print("Merge Sort:", merge_sorted)
```

```
Bubble Sort: [11, 12, 22, 25, 64]
Selection Sort: [11, 12, 22, 25, 64]
Insertion Sort: [11, 12, 22, 25, 64]
Quick Sort: [11, 12, 22, 25, 64]
Merge Sort: [11, 12, 22, 25, 64]
```

Write a Python program to search an element in an integer array using:Linear Search, Sentinel Search, Binary Search

In [25]:
```
def linear_search(arr, target):
    for i in range(len(arr)):
        if arr[i] == target:
            return i
    return -1

def sentinel_search(arr, target):
    n = len(arr)
    last_element = arr[n - 1]
    arr[n - 1] = target

    i = 0
    while arr[i] != target:
        i += 1

    arr[n - 1] = last_element

    if (i < n - 1) or (arr[n - 1] == target):
        return i
    return -1

def binary_search(arr, target):
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            low = mid + 1
        else:
            high = mid - 1

    return -1

# Example usage:
arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
target = 6

linear_search_result = linear_search(arr, target)
sentinel_search_result = sentinel_search(arr, target)
binary_search_result = binary_search(arr, target)

if linear_search_result != -1:
    print(f"Linear Search: Element {target} found at index {linear_search_re
else:
    print("Linear Search: Element not found")

if sentinel_search_result != -1:
    print(f"Sentinel Search: Element {target} found at index {sentinel_searc
else:
    print("Sentinel Search: Element not found")

if binary_search_result != -1:
    print(f"Binary Search: Element {target} found at index {binary_search_re
else:
    print("Binary Search: Element not found")
```

```
Linear Search: Element 6 found at index 5
Sentinel Search: Element 6 found at index 5
Binary Search: Element 6 found at index 5
```

Implement Graph in Python to perform following operations-Create, Adjacency Matrix, Adjacency List, Indegree, Outdegree

In [27]:
```
# Define a class for a directed graph
class Graph:
    def __init__(self, vertices):
        self.vertices = vertices
        self.adj_matrix = [[0] * vertices for _ in range(vertices)]  # Initi
        self.adj_list = {}  # Initialize an adjacency list

        for i in range(vertices):
            self.adj_list[i] = []  # Initialize the adjacency list for each

    def add_edge(self, start, end):
        if start < 0 or start >= self.vertices or end < 0 or end >= self.ver
            raise ValueError("Invalid vertex indices")  # Check if the verte

        self.adj_matrix[start][end] = 1  # Mark the edge in the adjacency ma
        self.adj_list[start].append(end)  # Add the end vertex to the adjace

    def adjacency_matrix(self):
        for row in self.adj_matrix:
            print(row)  # Display the adjacency matrix

    def adjacency_list(self):
        for vertex, neighbors in self.adj_list.items():
            print(f"{vertex} -> {', '.join(map(str, neighbors))}")  # Displa

    def indegree(self, vertex):
        if vertex < 0 or vertex >= self.vertices:
            raise ValueError("Invalid vertex index")  # Check if the vertex

        indeg = 0
        for i in range(self.vertices):
            indeg += self.adj_matrix[i][vertex]  # Count the incoming edges
        return indeg

    def outdegree(self, vertex):
        if vertex < 0 or vertex >= self.vertices:
            raise ValueError("Invalid vertex index")  # Check if the vertex

        outdeg = 0
        for i in range(self.vertices):
            outdeg += self.adj_matrix[vertex][i]  # Count the outgoing edges
        return outdeg

# Example usage:
num_vertices = 5
g = Graph(num_vertices)  # Create a graph with 5 vertices
g.add_edge(0, 1)  # Add edges between vertices
g.add_edge(0, 2)
g.add_edge(1, 2)
g.add_edge(2, 3)
g.add_edge(3, 0)
g.add_edge(3, 4)

print("Adjacency Matrix:")
g.adjacency_matrix()  # Display the adjacency matrix

print("\nAdjacency List:")
g.adjacency_list()  # Display the adjacency list

print("\nIndegree and Outdegree:")
for i in range(num_vertices):
```

```
    print(f"Vertex {i}: Indegree = {g.indegree(i)}, Outdegree = {g.outdegree
```

```
Adjacency Matrix:
[0, 1, 1, 0, 0]
[0, 0, 1, 0, 0]
[0, 0, 0, 1, 0]
[1, 0, 0, 0, 1]
[0, 0, 0, 0, 0]

Adjacency List:
0 -> 1, 2
1 -> 2
2 -> 3
3 -> 0, 4
4 ->

Indegree and Outdegree:
Vertex 0: Indegree = 1, Outdegree = 2
Vertex 1: Indegree = 1, Outdegree = 1
Vertex 2: Indegree = 2, Outdegree = 1
Vertex 3: Indegree = 1, Outdegree = 2
Vertex 4: Indegree = 1, Outdegree = 0
```

In [ ]: