In [1]:
```
#Write a program to implementCircular singly linked list with required data members and
#memberfunction(constructor, insertfirst, insertpos, insertlast, deletefirst, deletelas

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


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

    def insertfirst(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            new_node.next = self.head
        else:
            temp = self.head
            while temp.next != self.head:
                temp = temp.next
            temp.next = new_node
            new_node.next = self.head
            self.head = new_node

    def insertlast(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            new_node.next = self.head
        else:
            temp = self.head
            while temp.next != self.head:
                temp = temp.next
            temp.next = new_node
            new_node.next = self.head

    def insertpos(self, pos, data):
        if pos == 1:
            self.insertfirst(data)
            return
        new_node = Node(data)
        temp = self.head
        for i in range(pos - 2):
            temp = temp.next
            if temp == self.head:
                print("Position out of range")
                return
        new_node.next = temp.next
        temp.next = new_node

    def deletefirst(self):
        if self.head is None:
            print("List is empty")
            return
        if self.head.next == self.head:
            self.head = None
        else:
            temp = self.head
            while temp.next != self.head:
```

```
                temp = temp.next
            temp.next = self.head.next
            self.head = self.head.next

    def deletelast(self):
        if self.head is None:
            print("List is empty")
            return
        if self.head.next == self.head:
            self.head = None
        else:
            temp = self.head
            while temp.next.next != self.head:
                temp = temp.next
            temp.next = self.head

    def deletepos(self, pos):
        if self.head is None:
            print("List is empty")
            return
        if pos == 1:
            self.deletefirst()
            return
        temp = self.head
        for i in range(pos - 2):
            temp = temp.next
            if temp.next == self.head:
                print("Position out of range")
                return
        temp.next = temp.next.next

    def display(self):
        if self.head is None:
            print("List is empty")
            return
        temp = self.head
        while True:
            print(temp.data, end=" -> ")
            temp = temp.next
            if temp == self.head:
                break
        print("(head)")


cll = CircularSinglyLinkedList()
cll.insertlast(10)
cll.insertlast(20)
cll.insertlast(30)
cll.insertfirst(5)
cll.insertpos(3, 15)
cll.display()

cll.deletefirst()
cll.display()

cll.deletelast()
cll.display()

cll.deletepos(2)
cll.display()
```

```
5 -> 10 -> 15 -> 20 -> 30 -> (head)
10 -> 15 -> 20 -> 30 -> (head)
10 -> 15 -> 20 -> (head)
10 -> 20 -> (head)
```

In [2]:

```python
#Write a program to reverse Singly Linked List, Doubly Linked List
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None


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

    def insert(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
        else:
            temp = self.head
            while temp.next:
                temp = temp.next
            temp.next = new_node

    def reverse(self):
        prev = None
        current = self.head
        while current:
            nxt = current.next
            current.next = prev
            prev = current
            current = nxt
        self.head = prev

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


class DNode:
    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None


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

    def insert(self, data):
        new_node = DNode(data)
        if self.head is None:
            self.head = new_node
        else:
            temp = self.head
            while temp.next:
                temp = temp.next
```

```
            temp.next = new_node
            new_node.prev = temp

    def reverse(self):
        temp = None
        current = self.head
        while current:
            temp = current.prev
            current.prev = current.next
            current.next = temp
            current = current.prev
        if temp:
            self.head = temp.prev

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


sll = SinglyLinkedList()
sll.insert(1)
sll.insert(2)
sll.insert(3)
sll.insert(4)
print("Singly Linked List:")
sll.display()
sll.reverse()
print("Reversed Singly Linked List:")
sll.display()

dll = DoublyLinkedList()
dll.insert(10)
dll.insert(20)
dll.insert(30)
dll.insert(40)
print("Doubly Linked List:")
dll.display()
dll.reverse()
print("Reversed Doubly Linked List:")
dll.display()
```

```
Singly Linked List:
1 -> 2 -> 3 -> 4 -> None
Reversed Singly Linked List:
4 -> 3 -> 2 -> 1 -> None
Doubly Linked List:
10 <-> 20 <-> 30 <-> 40 <-> None
Reversed Doubly Linked List:
40 <-> 30 <-> 20 <-> 10 <-> None
```

In [3]:

```
# Write a program to implement STACK using Array with PUSH, POP operations
class Stack:
    def __init__(self, size):
        self.size = size
        self.stack = [None] * size
        self.top = -1

    def push(self, item):
        if self.top == self.size - 1:
            print("Stack Overflow")
```

```
        else:
            self.top += 1
            self.stack[self.top] = item
            print("Pushed", item)

    def pop(self):
        if self.top == -1:
            print("Stack Underflow")
        else:
            item = self.stack[self.top]
            self.stack[self.top] = None
            self.top -= 1
            print("Popped", item)

    def display(self):
        if self.top == -1:
            print("Stack is empty")
        else:
            print("Stack elements:", end=" ")
            for i in range(self.top + 1):
                print(self.stack[i], end=" ")
            print()


s = Stack(5)
s.push(10)
s.push(20)
s.push(30)
s.display()
s.pop()
s.display()
s.pop()
s.pop()
s.pop()
```

```
Pushed 10
Pushed 20
Pushed 30
Stack elements: 10 20 30
Popped 30
Stack elements: 10 20
Popped 20
Popped 10
Stack Underflow
```

In [4]:

```
#Write a program to implement Stack using Linked List with PUSH, POP,PEEK and Display o

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


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

    def push(self, data):
        new_node = Node(data)
        new_node.next = self.top
        self.top = new_node
        print("Pushed", data)
```

```
    def pop(self):
        if self.top is None:
            print("Stack Underflow")
        else:
            item = self.top.data
            self.top = self.top.next
            print("Popped", item)

    def peek(self):
        if self.top is None:
            print("Stack is empty")
        else:
            print("Top element is:", self.top.data)

    def display(self):
        if self.top is None:
            print("Stack is empty")
        else:
            temp = self.top
            print("Stack elements:", end=" ")
            while temp:
                print(temp.data, end=" ")
                temp = temp.next
            print()


s = Stack()
s.push(10)
s.push(20)
s.push(30)
s.display()
s.peek()
s.pop()
s.display()
s.peek()
```

```
Pushed 10
Pushed 20
Pushed 30
Stack elements: 30 20 10
Top element is: 30
Popped 30
Stack elements: 20 10
Top element is: 20
```

In [5]:
```
#Write a application of stack to Check for balanced parentheses.
class Stack:
    def __init__(self, size):
        self.size = size
        self.stack = [None] * size
        self.top = -1

    def push(self, item):
        if self.top == self.size - 1:
            return False
        self.top += 1
        self.stack[self.top] = item
        return True

    def pop(self):
        if self.top == -1:
            return None
        item = self.stack[self.top]
```

```
        self.stack[self.top] = None
        self.top -= 1
        return item

    def peek(self):
        if self.top == -1:
            return None
        return self.stack[self.top]

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


def is_balanced(expression):
    stack = Stack(len(expression))
    for char in expression:
        if char in "({[":
            stack.push(char)
        elif char in ")}]":
            if stack.is_empty():
                return False
            top = stack.pop()
            if (char == ")" and top != "(") or \
               (char == "}" and top != "{") or \
               (char == "]" and top != "["):
                return False
    return stack.is_empty()


expr1 = "{[()()]}"
expr2 = "{[(])}"
print(expr1, "-> Balanced?", is_balanced(expr1))
print(expr2, "-> Balanced?", is_balanced(expr2))
```

```
{[()()]} -> Balanced? True
{[(])} -> Balanced? False
```

In [6]:

```
# Write a program to Reverse a string using stack
class Stack:
    def __init__(self, size):
        self.size = size
        self.stack = [None] * size
        self.top = -1

    def push(self, item):
        if self.top == self.size - 1:
            return False
        self.top += 1
        self.stack[self.top] = item
        return True

    def pop(self):
        if self.top == -1:
            return None
        item = self.stack[self.top]
        self.stack[self.top] = None
        self.top -= 1
        return item

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

```
def reverse_string(s):
    stack = Stack(len(s))
    for char in s:
        stack.push(char)
    reversed_str = ""
    while not stack.is_empty():
        reversed_str += stack.pop()
    return reversed_str


string = "HELLO"
print("Original String:", string)
print("Reversed String:", reverse_string(string))
```

```
Original String: HELLO
Reversed String: OLLEH
```

In [7]:

```
#Write a program to implement  Linear Queue using list
class LinearQueue:
    def __init__(self, size):
        self.size = size
        self.queue = [None] * size
        self.front = -1
        self.rear = -1

    def enqueue(self, item):
        if self.rear == self.size - 1:
            print("Queue Overflow")
        else:
            if self.front == -1:
                self.front = 0
            self.rear += 1
            self.queue[self.rear] = item
            print("Enqueued", item)

    def dequeue(self):
        if self.front == -1 or self.front > self.rear:
            print("Queue Underflow")
        else:
            item = self.queue[self.front]
            self.queue[self.front] = None
            self.front += 1
            print("Dequeued", item)

    def display(self):
        if self.front == -1 or self.front > self.rear:
            print("Queue is empty")
        else:
            print("Queue elements:", end=" ")
            for i in range(self.front, self.rear + 1):
                print(self.queue[i], end=" ")
            print()


q = LinearQueue(5)
q.enqueue(10)
q.enqueue(20)
q.enqueue(30)
q.display()
q.dequeue()
q.display()
```

```
Enqueued 10
Enqueued 20
Enqueued 30
Queue elements: 10 20 30
Dequeued 10
Queue elements: 20 30
```

In [11]:

```python
# Write a program to Reverse stack using queue
class Stack:
    def __init__(self, size):
        self.size = size
        self.stack = [None] * size
        self.top = -1

    def push(self, item):
        if self.top == self.size - 1:
            print("Stack Overflow")
        else:
            self.top += 1
            self.stack[self.top] = item

    def pop(self):
        if self.top == -1:
            print("Stack Underflow")
            return None
        item = self.stack[self.top]
        self.stack[self.top] = None
        self.top -= 1
        return item

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

    def display(self):
        if self.top == -1:
            print("Stack is empty")
        else:
            print("Stack elements:", end=" ")
            for i in range(self.top + 1):
                print(self.stack[i], end=" ")
            print()


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

    def enqueue(self, item):
        if (self.rear + 1) % self.size == self.front:
            print("Queue Overflow")
        else:
            if self.front == -1:
                self.front = 0
            self.rear = (self.rear + 1) % self.size
            self.queue[self.rear] = item

    def dequeue(self):
        if self.front == -1:
            print("Queue Underflow")
            return None
```

```
        item = self.queue[self.front]
        self.queue[self.front] = None
        if self.front == self.rear:
            self.front = -1
            self.rear = -1
        else:
            self.front = (self.front + 1) % self.size
        return item

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


def reverse_stack(stack):
    q = Queue(stack.size)
    while not stack.is_empty():
        q.enqueue(stack.pop())
    while not q.is_empty():
        stack.push(q.dequeue())


s = Stack(5)
s.push(10)
s.push(20)
s.push(30)
s.push(40)
print("Original Stack:")
s.display()

reverse_stack(s)
print("Reversed Stack:")
s.display()
```

```
Original Stack:
Stack elements: 10 20 30 40
Reversed Stack:
Stack elements: 40 30 20 10
```

In [8]:

```
#write a Program to implement  Circular Queue using list

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

    def enqueue(self, item):
        if (self.rear + 1) % self.size == self.front:
            print("Queue Overflow")
        else:
            if self.front == -1:
                self.front = 0
            self.rear = (self.rear + 1) % self.size
            self.queue[self.rear] = item
            print("Enqueued", item)

    def dequeue(self):
        if self.front == -1:
            print("Queue Underflow")
        else:
            item = self.queue[self.front]
            self.queue[self.front] = None
```

```
            if self.front == self.rear:
                self.front = -1
                self.rear = -1
            else:
                self.front = (self.front + 1) % self.size
            print("Dequeued", item)

    def display(self):
        if self.front == -1:
            print("Queue is empty")
        else:
            print("Queue elements:", end=" ")
            i = self.front
            while True:
                print(self.queue[i], end=" ")
                if i == self.rear:
                    break
                i = (i + 1) % self.size
            print()


cq = CircularQueue(5)
cq.enqueue(10)
cq.enqueue(20)
cq.enqueue(30)
cq.display()
cq.dequeue()
cq.display()
cq.enqueue(40)
cq.enqueue(50)
cq.enqueue(60)  # should show overflow
cq.display()
```

```
Enqueued 10
Enqueued 20
Enqueued 30
Queue elements: 10 20 30
Dequeued 10
Queue elements: 20 30
Enqueued 40
Enqueued 50
Enqueued 60
Queue elements: 20 30 40 50 60
```

In [9]:

```
# Write a program to implement binary search tree with its operations
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None


class BST:
    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 search(self, key):
        return self._search(self.root, key)

    def _search(self, root, key):
        if root is None or root.key == key:
            return root
        if key < root.key:
            return self._search(root.left, key)
        return self._search(root.right, key)

    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
            temp = self._min_value_node(root.right)
            root.key = temp.key
            root.right = self._delete(root.right, temp.key)
        return root

    def _min_value_node(self, node):
        current = node
        while current.left is not None:
            current = current.left
        return current

    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=" ")



bst = BST()
bst.insert(50)
bst.insert(30)
bst.insert(70)
bst.insert(20)
bst.insert(40)
bst.insert(60)
bst.insert(80)

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

print("Preorder traversal:")
bst.preorder()

print("Postorder traversal:")
bst.postorder()

print("Search 40:", "Found" if bst.search(40) else "Not Found")
print("Search 90:", "Found" if bst.search(90) else "Not Found")

bst.delete(20)
print("Inorder after deleting 20:")
bst.inorder()

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

bst.delete(50)
print("Inorder after deleting 50:")
bst.inorder()
```

```
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
Search 40: Found
Search 90: Not Found
Inorder after deleting 20:
30 40 50 60 70 80
Inorder after deleting 30:
40 50 60 70 80
Inorder after deleting 50:
40 60 70 80
```

In [10]:

```
# Write a program to implement AVL Tree.

class Node:
    def __init__(self, key):
        self.key = key
```

```
        self.left = None
        self.right = None
        self.height = 1


class AVLTree:
    def get_height(self, root):
        if not root:
            return 0
        return root.height

    def get_balance(self, root):
        if not root:
            return 0
        return self.get_height(root.left) - self.get_height(root.right)

    def right_rotate(self, y):
        x = y.left
        T2 = x.right
        x.right = y
        y.left = T2
        y.height = 1 + max(self.get_height(y.left), self.get_height(y.right))
        x.height = 1 + max(self.get_height(x.left), self.get_height(x.right))
        return x

    def left_rotate(self, x):
        y = x.right
        T2 = y.left
        y.left = x
        x.right = T2
        x.height = 1 + max(self.get_height(x.left), self.get_height(x.right))
        y.height = 1 + max(self.get_height(y.left), self.get_height(y.right))
        return y

    def insert(self, root, key):
        if not root:
            return Node(key)
        elif key < root.key:
            root.left = self.insert(root.left, key)
        elif key > root.key:
            root.right = self.insert(root.right, key)
        else:
            return root

        root.height = 1 + max(self.get_height(root.left), self.get_height(root.right))
        balance = self.get_balance(root)

        if balance > 1 and key < root.left.key:
            return self.right_rotate(root)
        if balance < -1 and key > root.right.key:
            return self.left_rotate(root)
        if balance > 1 and key > root.left.key:
            root.left = self.left_rotate(root.left)
            return self.right_rotate(root)
        if balance < -1 and key < root.right.key:
            root.right = self.right_rotate(root.right)
            return self.left_rotate(root)

        return root

    def min_value_node(self, root):
        current = root
        while current.left:
```

```
            current = current.left
        return current

    def delete(self, root, key):
        if not root:
            return root
        elif key < root.key:
            root.left = self.delete(root.left, key)
        elif key > root.key:
            root.right = self.delete(root.right, key)
        else:
            if not root.left:
                return root.right
            elif not root.right:
                return root.left
            temp = self.min_value_node(root.right)
            root.key = temp.key
            root.right = self.delete(root.right, temp.key)

        root.height = 1 + max(self.get_height(root.left), self.get_height(root.right))
        balance = self.get_balance(root)

        if balance > 1 and self.get_balance(root.left) >= 0:
            return self.right_rotate(root)
        if balance > 1 and self.get_balance(root.left) < 0:
            root.left = self.left_rotate(root.left)
            return self.right_rotate(root)
        if balance < -1 and self.get_balance(root.right) <= 0:
            return self.left_rotate(root)
        if balance < -1 and self.get_balance(root.right) > 0:
            root.right = self.right_rotate(root.right)
            return self.left_rotate(root)

        return root

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

    def preorder(self, root):
        if root:
            print(root.key, end=" ")
            self.preorder(root.left)
            self.preorder(root.right)

    def postorder(self, root):
        if root:
            self.postorder(root.left)
            self.postorder(root.right)
            print(root.key, end=" ")



tree = AVLTree()
root = None
nums = [10, 20, 30, 40, 50, 25]

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

print("Inorder traversal of AVL tree:")
```

```
tree.inorder(root)
print("\nPreorder traversal of AVL tree:")
tree.preorder(root)
print("\nPostorder traversal of AVL tree:")
tree.postorder(root)

root = tree.delete(root, 40)
print("\n\nAfter deleting 40, inorder traversal:")
tree.inorder(root)
```

```
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
Search 40: Found
Search 90: Not Found
Inorder after deleting 20:
30 40 50 60 70 80
Inorder after deleting 30:
40 50 60 70 80
Inorder after deleting 50:
40 60 70 80
```

In [12]:

```
# Write a program to implement Graph
#1.Adjacency List
#2.Adjacency matrix
class GraphAdjList:
    def __init__(self, vertices):
        self.vertices = vertices
        self.graph = [[] for _ in range(vertices)]

    def add_edge(self, u, v):
        self.graph[u].append(v)
        self.graph[v].append(u)  # for undirected graph

    def display(self):
        print("Adjacency List:")
        for i in range(self.vertices):
            print(i, "->", self.graph[i])


class GraphAdjMatrix:
    def __init__(self, vertices):
        self.vertices = vertices
        self.graph = [[0] * vertices for _ in range(vertices)]

    def add_edge(self, u, v):
        self.graph[u][v] = 1
        self.graph[v][u] = 1  # for undirected graph

    def display(self):
        print("Adjacency Matrix:")
        for row in self.graph:
            print(row)



vertices = 5


g_list = GraphAdjList(vertices)
```

```
g_list.add_edge(0, 1)
g_list.add_edge(0, 4)
g_list.add_edge(1, 2)
g_list.add_edge(1, 3)
g_list.add_edge(1, 4)
g_list.add_edge(2, 3)
g_list.add_edge(3, 4)
g_list.display()

print()


g_matrix = GraphAdjMatrix(vertices)
g_matrix.add_edge(0, 1)
g_matrix.add_edge(0, 4)
g_matrix.add_edge(1, 2)
g_matrix.add_edge(1, 3)
g_matrix.add_edge(1, 4)
g_matrix.add_edge(2, 3)
g_matrix.add_edge(3, 4)
g_matrix.display()
```

```
Adjacency List:
0 -> [1, 4]
1 -> [0, 2, 3, 4]
2 -> [1, 3]
3 -> [1, 2, 4]
4 -> [0, 1, 3]

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

In [13]:

```
# Write a Program to find the element in an array using
# 1.Binary Search 2.Linear Search

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


def binary_search(arr, target):
    left = 0
    right = len(arr) - 1
    while left <= right:
        mid = (left + right) // 2
        if arr[mid] == target:
            return mid
        elif arr[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1



arr = [10, 20, 30, 40, 50, 60, 70]
target1 = 40
```

```
target2 = 25

print("Array:", arr)

index = linear_search(arr, target1)
if index != -1:
    print(f"Linear Search: Element {target1} found at index {index}")
else:
    print(f"Linear Search: Element {target1} not found")

index = binary_search(arr, target1)
if index != -1:
    print(f"Binary Search: Element {target1} found at index {index}")
else:
    print(f"Binary Search: Element {target1} not found")


index = linear_search(arr, target2)
print(f"Linear Search: Element {target2} {'found at index ' + str(index) if index != -1

index = binary_search(arr, target2)
print(f"Binary Search: Element {target2} {'found at index ' + str(index) if index != -1
```

```
Array: [10, 20, 30, 40, 50, 60, 70]
Linear Search: Element 40 found at index 3
Binary Search: Element 40 found at index 3
Linear Search: Element 25 not found
Binary Search: Element 25 not found
```

In [14]:

```
# Write a program to implement sorting using Divide and Conquer strategy
# 1. Quick sort 2.Merge sort
def quick_sort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)
        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


def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        L = arr[:mid]
        R = arr[mid:]

        merge_sort(L)
        merge_sort(R)

        i = j = k = 0
        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
```

```
            else:
                arr[k] = R[j]
                j += 1
            k += 1

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

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



arr1 = [34, 7, 23, 32, 5, 62]
arr2 = [34, 7, 23, 32, 5, 62]

print("Original Array for Quick Sort:", arr1)
quick_sort(arr1, 0, len(arr1) - 1)
print("Sorted Array using Quick Sort:", arr1)

print("\nOriginal Array for Merge Sort:", arr2)
merge_sort(arr2)
print("Sorted Array using Merge Sort:", arr2)
```

```
Original Array for Quick Sort: [34, 7, 23, 32, 5, 62]
Sorted Array using Quick Sort: [5, 7, 23, 32, 34, 62]

Original Array for Merge Sort: [34, 7, 23, 32, 5, 62]
Sorted Array using Merge Sort: [5, 7, 23, 32, 34, 62]
```

In [ ]: