LIST OF DATA STRUCTURES LAB PROGRAMS

1. Design and develop a python program to implement single linked list operations.
[Insertion at begining, at specific position, at ending]

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


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

    # Insert at beginning
    def insert_beginning(self, data):
        new_node = Node(data)
        new_node.next = self.head
        self.head = new_node

    # Insert at end
    def insert_end(self, data):
        new_node = Node(data)

        if self.head is None:
            self.head = new_node
            return

        temp = self.head
        while temp.next:
            temp = temp.next

        temp.next = new_node

    # Insert at specific position (1-based index)
    def insert_position(self, data, position):
        new_node = Node(data)

        if position == 1:
            new_node.next = self.head
            self.head = new_node
            return

        temp = self.head
        for i in range(position - 2):
            if temp is None:
                print("Position out of range")
```

```
            return
        temp = temp.next

    if temp is None:
        print("Position out of range")
        return

    new_node.next = temp.next
    temp.next = new_node

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


# Main Program
if __name__ == "__main__":
  ll = LinkedList()

  # Insert at beginning
  ll.insert_beginning(30)
  ll.insert_beginning(20)
  ll.insert_beginning(10)

  print("After insertion at beginning:")
  ll.display()

  # Insert at end
  ll.insert_end(40)
  ll.insert_end(50)

  print("After insertion at end:")
  ll.display()

  # Insert at specific position
  ll.insert_position(25, 3)

  print("After insertion at position 3:")
  ll.display()
```

OUTPUT :

```
After insertion at beginning:
10 -> 20 -> 30 -> None
After insertion at end:
10 -> 20 -> 30 -> 40 -> 50 -> None
After insertion at position 3:
10 -> 20 -> 25 -> 30 -> 40 -> 50 -> None
```

2. Design and develop a python program to implement double linked list operations.
[Deletion at begining, at specifc position, at ending]

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


# Doubly Linked List class
class DoublyLinkedList:
    def __init__(self):
        self.head = None

    # Insert at end (for initial creation)
    def insert_end(self, data):
        new_node = Node(data)

        if self.head is None:
            self.head = new_node
            return

        temp = self.head
        while temp.next:
            temp = temp.next

        temp.next = new_node
        new_node.prev = temp

    # Delete at beginning
    def delete_beginning(self):
        if self.head is None:
            print("List is empty")
            return

        self.head = self.head.next

        if self.head:
            self.head.prev = None

    # Delete at end
    def delete_end(self):
        if self.head is None:
            print("List is empty")
            return

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

```
            return

        temp = self.head
        while temp.next:
            temp = temp.next

        temp.prev.next = None

    # Delete at specific position (1-based index)
    def delete_position(self, position):
        if self.head is None:
            print("List is empty")
            return

        if position == 1:
            self.delete_beginning()
            return

        temp = self.head
        for i in range(position - 1):
            if temp is None:
                print("Position out of range")
                return
            temp = temp.next

        if temp is None:
            print("Position out of range")
            return

        if temp.next:
            temp.next.prev = temp.prev

        if temp.prev:
            temp.prev.next = temp.next

    # Display forward
    def display(self):
        temp = self.head
        result = ""

        while temp:
            result += str(temp.data) + " <-> "
            temp = temp.next

        result += "None"
        print(result)


# Main Program
if __name__ == "__main__":
```

```
dll = DoublyLinkedList()

# Creating list
dll.insert_end(10)
dll.insert_end(20)
dll.insert_end(30)
dll.insert_end(40)
dll.insert_end(50)

print("Original List:")
dll.display()

# Delete at beginning
dll.delete_beginning()
print("After deleting at beginning:")
dll.display()

# Delete at end
dll.delete_end()
print("After deleting at end:")
dll.display()

# Delete at specific position
dll.delete_position(2)
print("After deleting at position 2:")
dll.display()
```

OUTPUT :

```
Original List:
10 <-> 20 <-> 30 <-> 40 <-> 50 <-> None
After deleting at beginning:
20 <-> 30 <-> 40 <-> 50 <-> None
After deleting at end:
20 <-> 30 <-> 40 <-> None
After deleting at position 2:
20 <-> 40 <-> None
```

3. Design and develop a python program to implement circular linked list operations.
[Operations : Searching and traversing]

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


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

    # 🔶 Insert at end (helper for testing)
    def insert_end(self, data):
        new_node = Node(data)

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

        temp = self.head
        while temp.next != self.head:
            temp = temp.next

        temp.next = new_node
        new_node.next = self.head

    # 🔶 Traversal
    def traverse(self):
        if self.head is None:
            print("List is empty")
            return

        temp = self.head
        print("Traversal:", end=" ")

        while True:
            print(temp.data, end=" -> ")
            temp = temp.next
            if temp == self.head:
                break

        print("(back to head)")

    # 🔶 Searching
    def search(self, key):
        if self.head is None:
```

```
            print("List is empty")
            return

        temp = self.head
        position = 1

        while True:
            if temp.data == key:
                print(f"Element {key} found at position {position}")
                return
            temp = temp.next
            position += 1
            if temp == self.head:
                break

        print(f"Element {key} not found")


# 🔷 Driver Code
cll = CircularLinkedList()

cll.insert_end(10)
cll.insert_end(20)
cll.insert_end(30)
cll.insert_end(40)

cll.traverse()

cll.search(30)
cll.search(50)
```

OUTPUT :

```
Traversal: 10 -> 20 -> 30 -> 40 -> (back to head)
Element 30 found at position 3
Element 50 not found
```

4. Design and develop a python program to implement stack operations using single linked list. [Operations: Push and Pop]

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


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

    # 🔶 Push operation
    def push(self, data):
        new_node = Node(data)
        new_node.next = self.top
        self.top = new_node
        print(f"{data} pushed into stack")

    # 🔶 Pop operation
    def pop(self):
        if self.top is None:
            print("Stack Underflow")
            return

        popped = self.top.data
        self.top = self.top.next
        print(f"{popped} popped from stack")

    # 🔶 Peek operation
    def peek(self):
        if self.top is None:
            print("Stack is empty")
        else:
            print(f"Top element is {self.top.data}")

    # 🔶 Display stack
    def display(self):
        if self.top is None:
            print("Stack is empty")
            return

        temp = self.top
        print("Stack elements:")

        while temp:
            print(temp.data, end=" -> ")
            temp = temp.next
```

```
        print("None")


# 🔷 Driver Code
s = Stack()

s.push(10)
s.push(20)
s.push(30)

s.display()

s.peek()

s.pop()
s.display()
```

OUTPUT :

```
10 pushed into stack
20 pushed into stack
30 pushed into stack
Stack elements:
30 -> 20 -> 10 -> None
Top element is 30
30 popped from stack
Stack elements:
20 -> 10 -> None
```

5. Design and develop a python program to implement Queue operations using single linked list.
[Operations : Enqueue & Dequeue]

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


class Queue:
    def __init__(self):
        self.front = None
        self.rear = None

    # 🔶 Enqueue (Insert at rear)
    def enqueue(self, data):
        new_node = Node(data)

        if self.rear is None:
            self.front = self.rear = new_node
            print(f"{data} inserted into queue")
            return

        self.rear.next = new_node
        self.rear = new_node
        print(f"{data} inserted into queue")

    # 🔶 Dequeue (Delete from front)
    def dequeue(self):
        if self.front is None:
            print("Queue Underflow")
            return

        temp = self.front
        print(f"{temp.data} removed from queue")
        self.front = self.front.next

        if self.front is None:
            self.rear = None

    # 🔶 Peek (Front element)
    def peek(self):
        if self.front is None:
            print("Queue is empty")
        else:
            print(f"Front element is {self.front.data}")

    # 🔶 Display queue
```

```
    def display(self):
        if self.front is None:
            print("Queue is empty")
            return

        temp = self.front
        print("Queue elements:")

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


# 🔷 Driver Code
q = Queue()

q.enqueue(10)
q.enqueue(20)
q.enqueue(30)

q.display()

q.peek()

q.dequeue()
q.display()
```

OUTPUT :

```
10 inserted into queue
20 inserted into queue
30 inserted into queue
Queue elements:
10 -> 20 -> 30 -> None
Front element is 10
10 removed from queue
Queue elements:
20 -> 30 -> None
```

6. Design and develop a python program to implement for conversion expression from infix to postfix and infix to prefix.

```
# Function to define precedence
def precedence(op):
    if op == '+' or op == '-':
        return 1
    elif op == '*' or op == '/':
        return 2
    elif op == '^':
        return 3
    return 0


# 🔶 Infix to Postfix
def infix_to_postfix(expression):
    stack = []
    postfix = ""

    for char in expression:
        # Operand
        if char.isalnum():
            postfix += char

        # Opening bracket
        elif char == '(':
            stack.append(char)

        # Closing bracket
        elif char == ')':
            while stack and stack[-1] != '(':
                postfix += stack.pop()
            stack.pop()  # remove '('

        # Operator
        else:
            while (stack and precedence(stack[-1]) >= precedence(char)):
                postfix += stack.pop()
            stack.append(char)

    # Pop remaining operators
    while stack:
        postfix += stack.pop()

    return postfix


# 🔶 Infix to Prefix
def infix_to_prefix(expression):
```

```
    # Reverse expression
    expression = expression[::-1]

    # Swap brackets
    new_expr = ""
    for char in expression:
        if char == '(':
            new_expr += ')'
        elif char == ')':
            new_expr += '('
        else:
            new_expr += char

    # Convert to postfix
    postfix = infix_to_postfix(new_expr)

    # Reverse postfix → prefix
    prefix = postfix[::-1]

    return prefix


# 🔷 Driver Code
expr = "A+B*(C-D)"

print("Infix Expression:", expr)

postfix = infix_to_postfix(expr)
print("Postfix Expression:", postfix)

prefix = infix_to_prefix(expr)
print("Prefix Expression:", prefix)
```

OUTPUT :

```
Infix Expression: A+B*(C-D)
Postfix Expression: ABCD-*+
Prefix Expression: +A*B-CD
```

7. Design and develop a Python program to implement Binary tree traversals.
[Operations :In-order, pre-order,post-order]

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


# 🔶 Inorder Traversal (Left → Root → Right)
def inorder(root):
   if root:
      inorder(root.left)
      print(root.data, end=" ")
      inorder(root.right)


# 🔶 Preorder Traversal (Root → Left → Right)
def preorder(root):
   if root:
      print(root.data, end=" ")
      preorder(root.left)
      preorder(root.right)


# 🔶 Postorder Traversal (Left → Right → Root)
def postorder(root):
   if root:
      postorder(root.left)
      postorder(root.right)
      print(root.data, end=" ")


# 🔷 Creating Binary Tree manually
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)

# 🔷 Traversals
print("Inorder Traversal:")
inorder(root)

print("\nPreorder Traversal:")
preorder(root)

print("\nPostorder Traversal:")
```

postorder(root)
OUTPUT :

```
Inorder Traversal:
4 2 5 1 3
Preorder Traversal:
1 2 4 5 3
Postorder Traversal:
4 5 2 3 1
```

8. Design and develop a python program to implement graph traversals techniques.
[Operations : BFS & DFS]

```
from collections import deque

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

    # ◆ Add edge
    def add_edge(self, u, v):
        if u not in self.graph:
            self.graph[u] = []
        if v not in self.graph:
            self.graph[v] = []
        self.graph[u].append(v)
        self.graph[v].append(u)   # Remove this line for directed graph

    # ◆ BFS Traversal
    def bfs(self, start):
        visited = set()
        queue = deque([start])

        print("BFS Traversal:", end=" ")

        while queue:
            node = queue.popleft()

            if node not in visited:
                print(node, end=" ")
                visited.add(node)

                for neighbor in self.graph[node]:
                    if neighbor not in visited:
                        queue.append(neighbor)

    # ◆ DFS Traversal (Recursive)
    def dfs(self, node, visited=None):
        if visited is None:
            visited = set()

        if node not in visited:
            print(node, end=" ")
            visited.add(node)

            for neighbor in self.graph[node]:
                self.dfs(neighbor, visited)
```

```
# 🔷 Driver Code
g = Graph()

# Creating graph
g.add_edge(1, 2)
g.add_edge(1, 3)
g.add_edge(2, 4)
g.add_edge(2, 5)
g.add_edge(3, 6)

# 🔷 Traversals
print("DFS Traversal:", end=" ")
g.dfs(1)

print("\nBFS Traversal:", end=" ")
g.bfs(1)
```

OUTPUT :

```
DFS Traversal: 1 2 4 5 3 6
BFS Traversal: BFS Traversal: 1 2 3 4 5 6
```

9. Design and develop a python program to implement Binary search tree operations.
[Operations : In-order,pre-order,post-order]

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


class BST:
    # 🔶 Insert
    def insert(self, root, key):
        if root is None:
            return Node(key)

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

        return root

    # 🔶 Search
    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)

    # 🔶 Find minimum value (used in deletion)
    def min_value(self, node):
        current = node
        while current.left:
            current = current.left
        return current

    # 🔶 Delete
    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:
            # Node with one or no child
            if root.left is None:
                return root.right
            elif root.right is None:
                return root.left

            # Node with two children
            temp = self.min_value(root.right)
            root.key = temp.key
            root.right = self.delete(root.right, temp.key)

        return root

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

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

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


# 🔷 Driver Code
bst = BST()
root = None

# Insert elements
elements = [50, 30, 20, 40, 70, 60, 80]
for ele in elements:
    root = bst.insert(root, ele)

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

```
print("\nPreorder Traversal:")
bst.preorder(root)

print("\nPostorder Traversal:")
bst.postorder(root)

# Search
key = 40
if bst.search(root, key):
    print(f"\nElement {key} found")
else:
    print(f"\nElement {key} not found")

# Delete
root = bst.delete(root, 20)
print("After deleting 20 (Inorder):")
bst.inorder(root)
```

OUTPUT :

```
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
Element 40 found
After deleting 20 (Inorder):
30 40 50 60 70 80
```

10. Design and develop a python program to implement AVL trees.

```
class Node:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None
        self.height = 1


class AVLTree:

    # 🔶 Get height
    def get_height(self, root):
        if not root:
            return 0
        return root.height

    # 🔶 Get balance factor
    def get_balance(self, root):
        if not root:
            return 0
        return self.get_height(root.left) - self.get_height(root.right)

    # 🔶 Right Rotation (LL case)
    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

    # 🔶 Left Rotation (RR case)
    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

# ◆ Insert
def insert(self, root, key):
    # Step 1: Normal BST insert
    if not root:
        return Node(key)

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

    # Step 2: Update height
    root.height = 1 + max(self.get_height(root.left),
                          self.get_height(root.right))

    # Step 3: Get balance factor
    balance = self.get_balance(root)

    # Step 4: Balance the tree

    # LL Case
    if balance > 1 and key < root.left.key:
        return self.right_rotate(root)

    # RR Case
    if balance < -1 and key > root.right.key:
        return self.left_rotate(root)

    # LR Case
    if balance > 1 and key > root.left.key:
        root.left = self.left_rotate(root.left)
        return self.right_rotate(root)

    # RL Case
    if balance < -1 and key < root.right.key:
        root.right = self.right_rotate(root.right)
        return self.left_rotate(root)

    return root

# ◆ Inorder Traversal
def inorder(self, root):
    if root:
        self.inorder(root.left)
        print(root.key, end=" ")
        self.inorder(root.right)
```

```
# 🔷 Driver Code
avl = AVLTree()
root = None

elements = [10, 20, 30, 40, 50, 25]

for ele in elements:
    root = avl.insert(root, ele)

print("Inorder Traversal of AVL Tree:")
avl.inorder(root)
```

OUPUT :

```
Inorder Traversal of AVL Tree:
10 20 25 30 40 50
```

11. Design and develop a python program to implement Quick sort and Insertion sort.

```
# 🔶 Quick Sort
def quick_sort(arr):
    if len(arr) <= 1:
        return arr

    pivot = arr[0]
    left = [x for x in arr[1:] if x <= pivot]
    right = [x for x in arr[1:] if x > pivot]

    return quick_sort(left) + [pivot] + quick_sort(right)


# 🔶 Insertion Sort
def insertion_sort(arr):
    for i in range(1, len(arr)):
        key = arr[i]
        j = i - 1

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

        arr[j + 1] = key

    return arr


# 🔷 Driver Code
arr1 = [64, 34, 25, 12, 22, 11, 90]
arr2 = arr1.copy()

print("Original Array:", arr1)

# Quick Sort
sorted_qs = quick_sort(arr1)
print("Sorted using Quick Sort:", sorted_qs)

# Insertion Sort
sorted_is = insertion_sort(arr2)
print("Sorted using Insertion Sort:", sorted_is)
```

OUTPUT :

```
Original Array: [64, 34, 25, 12, 22, 11, 90]
Sorted using Quick Sort: [11, 12, 22, 25, 34, 64, 90]
Sorted using Insertion Sort: [11, 12, 22, 25, 34, 64, 90]
```

12. Design and develop a python program to implement Heap sort and Merge sort.

```
# ◆ Heap Sort

def heapify(arr, n, i):
    largest = i
    left = 2 * i + 1
    right = 2 * i + 2

    # Check left child
    if left < n and arr[left] > arr[largest]:
        largest = left

    # Check right child
    if right < n and arr[right] > arr[largest]:
        largest = right

    # Swap and continue heapifying
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)


def heap_sort(arr):
    n = len(arr)

    # Build max heap
    for i in range(n // 2 - 1, -1, -1):
        heapify(arr, n, i)

    # Extract elements one by one
    for i in range(n - 1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)

    return arr


# ◆ Merge Sort

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

        # Recursively sort both halves
        merge_sort(left)
        merge_sort(right)
```

```
    i = j = k = 0

    # Merge two halves
    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            arr[k] = left[i]
            i += 1
        else:
            arr[k] = right[j]
            j += 1
        k += 1

    # Remaining elements
    while i < len(left):
        arr[k] = left[i]
        i += 1
        k += 1

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

  return arr


# 🔷 Driver Code
arr1 = [12, 11, 13, 5, 6, 7]
arr2 = arr1.copy()

print("Original Array:", arr1)

# Heap Sort
heap_sorted = heap_sort(arr1)
print("Sorted using Heap Sort:", heap_sorted)

# Merge Sort
merge_sorted = merge_sort(arr2)
print("Sorted using Merge Sort:", merge_sorted)
```

OUTPUT :

```
Original Array: [12, 11, 13, 5, 6, 7]
Sorted using Heap Sort: [5, 6, 7, 11, 12, 13]
Sorted using Merge Sort: [5, 6, 7, 11, 12, 13]
```

13. Design and develop a python program to implement linear search and Binary search.

```
# 🔶 Linear Search
def linear_search(arr, key):
    for i in range(len(arr)):
        if arr[i] == key:
            return i
    return -1


# 🔶 Binary Search (Iterative)
def binary_search(arr, key):
    low = 0
    high = len(arr) - 1

    while low <= high:
        mid = (low + high) // 2

        if arr[mid] == key:
            return mid
        elif arr[mid] < key:
            low = mid + 1
        else:
            high = mid - 1

    return -1


# 🔷 Driver Code
arr = [10, 20, 30, 40, 50]

key = 30

# Linear Search
result1 = linear_search(arr, key)
if result1 != -1:
    print(f"Element found at index {result1} using Linear Search")
else:
    print("Element not found using Linear Search")

# Binary Search (array must be sorted)
result2 = binary_search(arr, key)
if result2 != -1:
    print(f"Element found at index {result2} using Binary Search")
else:
    print("Element not found using Binary Search")
```

OUTPUT :

```
Element found at index 2 using Linear Search
Element found at index 2 using Binary Search
```