GATE INSTITUTE OF TECHNOLOGY & MANAGEMENT SCIENCES

(Approved by AICTE , Affiliated to Sri Venkateswara University , TIRUPATI)

ICET CODE : GTMT

Master of Computer Applications

SEMESTER - II

Hall Ticket No: 122531002

Name of the Student: A. Guru prasad

Subject: Data structures

Year: I st Semester: II Batch 2025-26

GATE EDUCATIONAL INSTITUTIONS

DEPARTMENT OF COMPUTERS

YEAR I st SEMESTER II

Reg No. 122531002

CERTIFICATE

Certified that this is the bonafide record of practical work done in the laboratory by the candidate A. Guru prasad *during the academic year* 2025-27 *subject* Data structures .

No. of Practical's Conducted 13 ***No. of Practical's Attended*** 13

LECTURER **HEAD OF THE DEPARTMENT**

Submitted for the Practical Examination held on _____

Examiners

1.

2.

INDEX

S.No.	Date	Name of the Experiment	Page No.	Marks Awarded	Remarks
01	18/03/26	Singly linked list - Insertion	1-5		
02	24/03/26	Doubly linked list - Deletion	6-11		
03	30/03/26	Circular linked list - Searching	12-16		
04	02/04/26	Stack using linked list	17-21		
05	07/04/26	Queue using linked list	22-26		
06	15/04/26	Expression conversion	27-31		
07	18/04/26	Binary Tree Traversal's	32-34		
08	22/04/26	Graph Traversals (BFS & DFS)	35-39		
09	27/04/26	Binary Search Tree (BST)	40-45		
10	30/04/26	AVL Tree	46-51		
11	02/06/26	Quick Sort & Insertion Sort	52-55		
12	05/06/26	Heap Sort & Merge Sort	56-60		
13	10/06/26	Linear & Binary Search	61-64		

INDEX

S.No.	Date	Name of the Experiment	Page No.	Marks Awarded	Remarks

Date :	GATE Educational Institutions	Page No. : 1
18/03/26	Singly Linked List - Insertion	Practical No. : 1

Aim :

To Perform insertion operations.

Description:

A singly linked list is a dynamic linear data structure where each node contains data and a reference to the next node. It does not store elements in contiguous memory like arrays. This allows efficient insertion and deletion operations without shifting elements. in Contiguous memory like arrays. This allows efficient insertion and deletion operations without shifting elements. However, traversal is only Possible in one direction.

Pesudocode:

```
START
Create Node with data and next pointer
Create LinkedList with head = NULL
INSERT_BEGINNING (data)
  Create new node
  new node next = head
  head = new node
END
  INSERT_END (data)
    Create new node
    IF head is NULL
        head = new node
    ELSE
        temp = head
        WHILE temp next is not NULL
```

```
      move temp to next node
    END WHILE
    temp next = new node
  END IF
END

INSERT_POSITION
  Create new node
  IF position=1
    newnode next = head
    head = new node
  ELSE
    temp = head
    Move temp to (Position -1) node
    IF Position invalid.
      Print "Position out of range"
    ELSE
      new node next = temp next
        temp next = new node
      END IF
    ENDIF
END
DISPLAY
  temp = head
  WHILE temp is not NULL
  Print temp data
  temp = temp next
  END WHILE
END
STOP
```

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
```

Doubly Linked List - Deletion

Aim :

To Perform deletion operations.

Description :

A doubly linked linked list is an extension of singly linked list where each node contains two pointers : one pointing to the previous node and one to the next node. This allows traversal in both forward and backward direction, making deletion operations easier compared to singly linked lists.

Pesudocode :

```
START
Create Node with:
    data
    Prev Pointer
    next Pointer
Create Doubly Linked List with head = NULL
DELETE_BEGINNING
   IF head is NULL
     Print "List is empty"
   ELSE
     head = head next
     IF head is not NULL
        head prev = NULL
     END IF
   END IF
END
```

```
DELETE_END
   IF head is NULL
      Print "List is empty"
   ELSE IF only one node exists
      head = NULL
   ELSE
      temp = head
      Move temp to last node
      temp prev next = NULL
   END IF
END
   DELETE_POSITION
    IF list isempty
      Print "List is empty"
    ELSE IF Position = 1
       DELETE_BEGINNING
    ELSE
    Move temp to given position
    IF Position invalid
      Print "Position out of range"
    ELSE
      temp prev next = temp next
       IF temp next exists
          temp next prev = temp prev
       END IF
   END IF
 END IF
END
   Display list  Stop
```

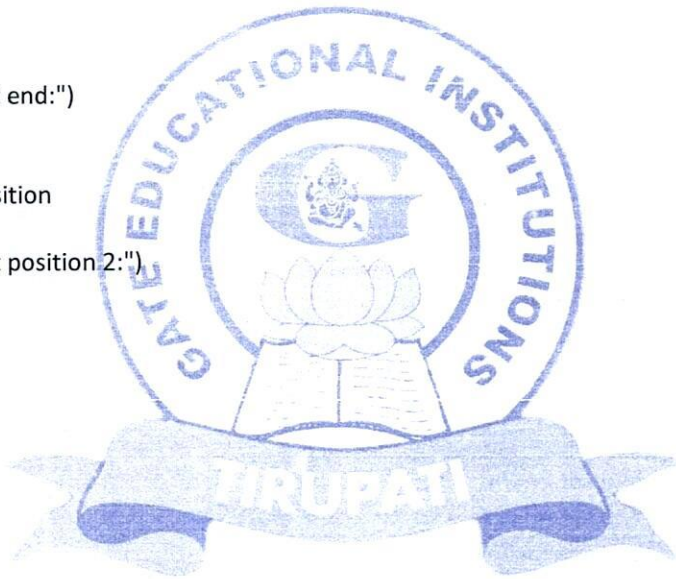
2. Design and develop a python program to implement double linked list operations.
[Deletion at begining, at specifc position, at ending]

```
# 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
```

Circular Linked List - Searching

Aim:

To search element

Description:

In a circular linked list, the last node connects back to the first node insted of Pointing to NULL This structure is useful in applications requiring a continuous loop, such as round-robin Scheduling.

Pesudocode:

```
START
Create Node with data and next pointer
Create Circular Linked List with head = NULL
INSERT_END
  Create new node
  IF head IS NULL
     head = new node
     new node next = head
  ELSE
     temp = head
      WHILE temp next != head
      temp = temp next
    ENDWHILE
      temp next = new node
      new node next = head
     END IF
 END
 TRAVERSE
   IF head is NULL
       Print "List is empty"
```

```
ELSE
   temp = head
   REPEAT
    Print temp data
    temp = temp next
   UNTIL temp = head
   ENDIF
END
SEARCH (key)
  IF head IS NULL
    Print "List is empty"
  ELSE
     temp = head
     Position = 1
     REPEAT
        IF temp data = key
           Print element found
           STOP
         END IF
        temp = temp next
        Position = position + 1
     UNTIL temp = head
     Print " Element not found"
  END IF
END
STOP.
```

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
```

Stack using Linked List

Aim:

To implement stack

Description:

A stack is a linear data structure that follows the LIFO (Last in First out) Principle. using linked list avoids Fixed size limitations and allows dynamic memory usage.

Pesudocode:

```
START
Create Node with data and next
Create stack with top = null
PUSH (data)
    Create new node
    new node next = top
    top = new node
END
POP
 IF top IS NULL
    Print "Stack underflow"
 ELSE
    temp = top
    top = top Next
    Delete temp
 ENDIF
END
PEEK
 IF top is NULL
    Print "stack is empty"
 ELSE
    Print top data
 ENDIF
```

```
END
DISPLAY
   temp = top
   WHILE temp is not NULL
   Print temp data
      temp = temp next
   END WHILE
END
STOP
```

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
```

Queue using Linked List

Aim:

To implement queue.

Description:

A queue is a FIFO (First in First out) data structure where insertion happens at the rear and deletion at the front. Linked list implementation avoids overflow limitations of arrays.

Pesudocode:

```
START
Create Node with data and next
create Queue with
    Front = NULL
    rear = NULL
ENQUEUE (data)
    Create new node
    IF rear IS NULL
      Front = rear = newnode
    ELSE
       rear next = new Node
        rear = new node
     END IF
END
DEQUEUE
    IF front IS NULL
      Print "Queue Under Flow"
```

```
    ELSE
        temp = front
        front = front next
        IF front IS NULL
            rear = NULL
        END IF
        Delete temp
        END IF
    END
    PEEK
      IF front is NULL
        Print "Queue is empty"
      ELSE
        Print front data
      END IF
END
DISPLAY queue
STOP
```

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
```

Expression Conversion

Aim:

Infix to Postfix / Prefix

Description:

Infix expressions are human-readable but require precedence rules. Postfix and Prefix notations eliminate ambiguity and are used in compilers and expression evaluation.

Pesudocode:

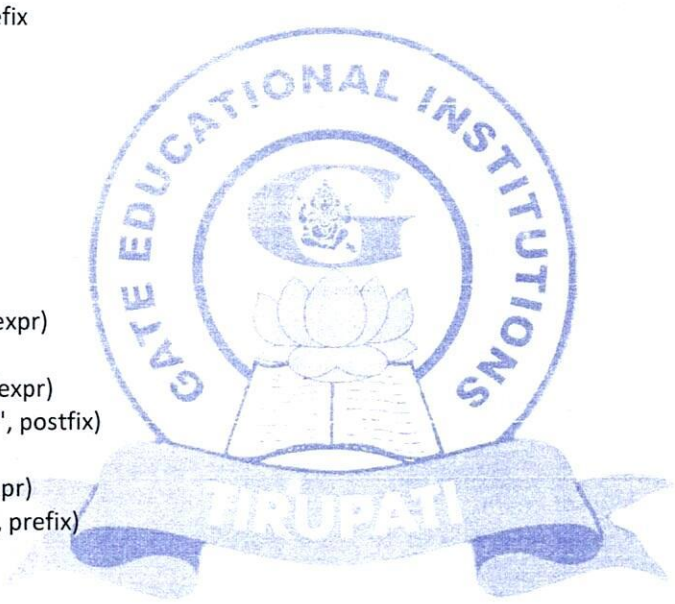
```
START
PRECEDENCE (operator)
   Return priority of operator
END
INFIX_TO_POSTFIX (expression)
Create empty stack
PostFix = " "
FOR each character in expression
   IF operand
     Add to PostFix
   ELSE IF '('
     Push into stack
   ELSE IF ')'
     POP until '(' found
   ELSE
     WHILE stack not empty AND
      precedence (top) >= precedence (current)
```

```
    PoP and add to Post Fix
  END WHILE
  Push operator
 END IF
END FOR
PoP remaining operators
RETURN Post Fix
END
INFIX_To_PREFIX (expression)
   Reverse expression
   Swap brackets
   Convert to Postfix
   Reverse Postfix result
  END
STOP
```

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
```

Binary Tree Traversals

Aim:

To Perform traversal

Description:

A binary tree is a hierarchical structure where each node has at most two children. Traversal method help visit all systematically for processing data.

Pesudocode:

```
START
Create Node with:
   data
   left child
   right child
INORDER (root)
   IF root exists
     INORDER (left)
     Print order data
     INORDER (right)
   END IF
END
Preorder (root)
     IF root exists
     Print root data
      PREORDER (left)
      PREORDER (Right)
   END IF
END
POSTORDER (root)
   IF root exists
      POSTORDER (left) (Right)
      Print root data
   End IF
End
   Stop.
```

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
```

Graph Traversals (BFS & DFS)

Aim :-

To traverse graph

Description :-

A graph consists of vertices and edges. BFS explores level using a queue, while DFS explores depth-wise using recursion or Stack.

Pesudocode :

```
START
Create Graph using adjacency list
ADD_EDGE (u,v)
    Add v to u list
    Add u to v list
END
BFS (start)
  Create visited set
  Create queue
  Insert start into queue
  WHILE queue not empty
    Remove node from queue
    IF node not visited
      Print node
      Mark visited
      Add all unvisited neighbors into queue
```

```
        ENDIF
    END WHILE
END

DFS (node)
IF node not visited
        Print node
        Mark visited
        FOR each neighbor
            DFS (neighbor)
        END FOR
    END IF

END

STOP
```

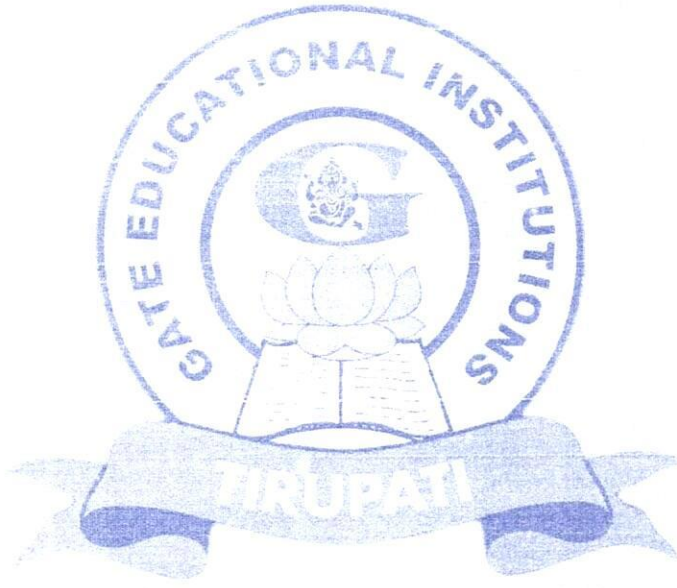
8. Design and develop a python program to implement graph traversals techniques.
[Operations : BFS & DFS]

```
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
```



Binary Search Tree (BST)

Aim:

To Perform operations.

Description:

A Binary search Tree is a tree where left subtree contains smaller values and right subtree contains larger values. It provides efficients searching, insertion and deletion.

Pesudocode:

```
START
Create Node with:
    key
    left child
    right child
INSERT (root, key)
    IF root is NULL
        Create node
    ELSE IF key < root key
        Insert into left subtree
    ELSE
        Insert into right subtree
    END IF
        Return root
END
```

```
SEARCH (root, key)
   IF root is NULL OR key found.
      Return root
   ELSE IF key < root key
      Search left subtree
   ELSE
      search right subtree

END
DELETE (root, key)
    Find node
    IF node has no child
      Delete node
     ELSE IF one child
    1 Replace with child
    ELSE
       Find inorder successor
       Replace node value
       Delete successor
    END IF
END
```

INORDER / PREORDER / POSTORDER traversal

Stop.

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
```

AVL Tree

Aim :

To maintain balance

Description :

AVL tree maintains balance by ensuring height difference between left and right subtrees is at most 1. Rotations are used to maintain balance after insertion.

Pesudo Code :

```
START
Create Node with :
    Key
    left
    right
    height
GET_HEIGHT (node)
    Return Node height
END
GET_BALANCE (node)
    Return left height - right height
END
RIGHT_ROTATE (node)
    Perform right rotation
END
LEFT_ROTATE (node)
    Perform left rotation
```

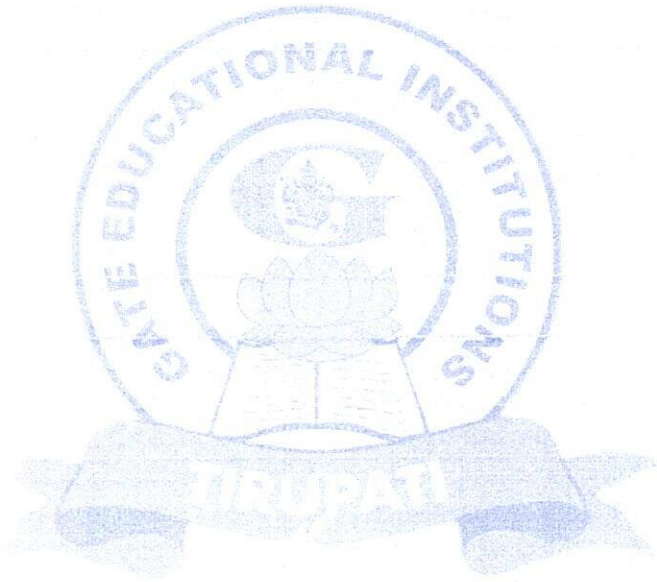
```
END
INSERT (root, key)
   Perform normal BST insertion
-update height
  Calculate balance factor
   IF LL imbalance
        Right rotate
    ELSE IF RR imbalance
        Left rotation
    ELSE IF LR imbalance
        Left rotate left child
        Right rotate root
    ELSE IF RL imbalance
        Right rotate left child
        Left rotate root
     END IF
       Return root
END

INORDER traversal

 STOP
```

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))
```

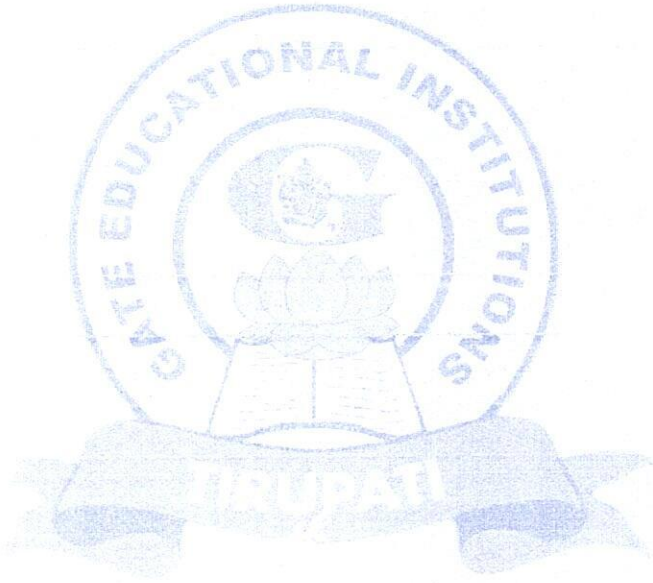
```
# ◇ Insert
def insert(self, root, key):
    # Step 1: Normal BST insert
    if not root:
        return Node(key)

    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
```



Quick Sort & Insertion Sort

Aim :

Sorting

Description:

Sorting is essential for efficien searching and data organization. Quick Sort uses divide-and-Conquer strategy, while insertion sort is simple and efficient for small data sets.

Pesudo Code:

```
START
QUICK_SORT (arry)
    IF array Size <=1
       Return array
    Choose Pivot
    Create left array with smaller elements
    Create right array with greater elements
     Return Quick_SORT (left) + Pivot + Quick_SORT(right)
  END
INSERTION_SORT (array)
   FOR i=1 to n-1
      key = array[i]
      j=i-1
      WHILE j>=0 AND array [j] > key
       Shift array [j]
```

```
        j = j-1
    ENDWHILE
    Insert key at correct position
END FOR
  Return array
END
STOP
```

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]
```

Date:	GATE Educational Institutions	Page No.: 56
		Practical No.:
05/06/26	Heap Sort & Merge Sort	12

Aim:

Efficient Sorting

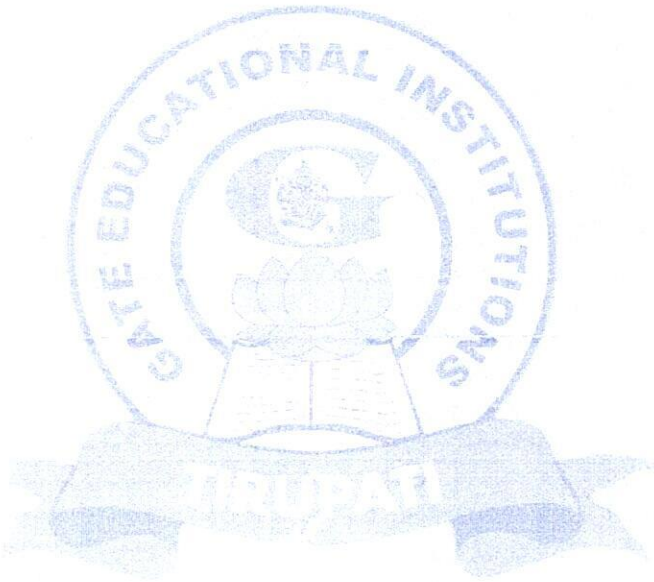
Description:

Heap Sort uses a binary heap structure, while merge sort divides the array into halves and merges them after sorting. Both are efficient for large datasets.

Pesudocode:

```
START
HEAPIFY (array, n, i)
    largest = i;
    Compare left child
    Compare right child
    IF largest changes
        Swap elements
        HEAPIFY again
    END IF
END
HEAP_SORT (array)
    Build max heap
    FOR each element from end
        Swap first and last
        Heapify reduced heap
    END FOR
END
```

```
MERGE_SORT (array)
    IF array size >1
      Divide array into two halves
      MERGE_SORT (left)
      MERGE_SORT (right)

      Merge Sorted halves
    END IF
END
STOP
```



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]
```

Linear & Binary search

Aim:

Searching techniques.

Description:

Searching helps locate elements in a dataset. Linear search scans sequentially, while binary search works on sorted arrays and reduces search space efficiently.

Pesudo code:

```
START
LINEAR_SEARCH (array, key)
    FOR each element
        IF element = key
            Return index
        END IF
    END FOR
    Return -1
END
BINARY_SEARCH (array, key)
      low = 0
      high = n-1
      WHILE low <= high
       mid = (low + high)/2
       IF array [mid] = key
        Return mid
      ELSE IF array [mid] < key
```

```
        low = mid + 1
    ELSE
        hight = mid - 1
    END IF
  END WHILE
  Return -1
END

STOP
```

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
```