**Practical 1: Traversing 1D array**

**Practical 2: Insertion of an element in 1D array**

**Practical 3: Deletion of an element in 1D array**

**Practical 4: Searching of an element in 1D array**

**Practical 5: Updating an element in 1D array**

**(Practical 1-5, DONE in practical session, code in c++)**

**Practical 6A: Traversal of linked list (insert and print) in python.**

# A single node of a singly linked list

class Node:

  # constructor

  def \_\_init\_\_(self, data = None, next=None):

    self.data = data

    self.next = next

# A Linked List class with a single head node

class LinkedList:

  def \_\_init\_\_(self):

    self.head = None

  # insertion method for the linked list

  def insert(self, data):

    newNode = Node(data)

    if(self.head):

      current = self.head

      while(current.next):

        current = current.next

      current.next = newNode

    else:

      self.head = newNode

  # print method for the linked list

  def printLL(self):

    current = self.head

    while(current):

      print(current.data)

      current = current.next

# Singly Linked List with insertion and print methods

LL = LinkedList()

LL.insert(3)

LL.insert(4)

LL.insert(5)

LL.printLL()

**Practical 6B: Insert at the end of the linked list.**

class Node:

def \_\_init\_\_(self,data):

self.data = data

self.next = None

class LinkedList:

def \_\_init\_\_(self):

self.head = None

def insertAtBeginning(self,new\_value):

new\_node = Node(new\_value)

new\_node.next = self.head

self.head = new\_node

print("New node inserted with data",new\_value)

lList = LinkedList()

lList.insertAtBeginning(5)

lList.insertAtBeginning(7)

**Practical 6C: Insert at the end of the linked list.**

class Node:

  def \_\_init\_\_(self,data):

    self.data = data

    self.next = None

class LinkedList:

  def \_\_init\_\_(self):

    self.head = None

  def insertAtEnd(self,new\_value):

    new\_node = Node(new\_value)

    if self.head is None:

      self.head = new\_node

      print("New node inserted with data",new\_value)

      return

    last\_node = self.head

    while(last\_node.next):

      last\_node = last\_node.next

    last\_node.next = new\_node # type: ignore

    print("New node inserted with data",new\_value)

lList = LinkedList()

lList.insertAtEnd(2)

lList.insertAtEnd(4)

**Practical 7A: Stack creation, push and pop operation**

# Stack Creation

def create\_stack():

stack = list() #declaring an empty list

return stack

# Checking for empty stack

def Isempty(stack):

return len(stack) == 0

# Inserting items into the stack

def push(stack, n):

stack.append(n)

print("pushed item: " + n)

# Removal of an element from the stack

def pop(stack):

if (Isempty(stack)):

return "stack is empty"

else:

return stack.pop()

# Displaying the stack elements

def show(stack):

print("The stack elements are:")

for i in stack:

print(i)

stack = create\_stack()

push(stack, str(10))

push(stack, str(20))

push(stack, str(30))

push(stack, str(40))

print("popped item: " + pop(stack))

show(stack)

**Practical 7B: Queue (enqueue and dequeue)**

class Queue:

def \_\_init\_\_(self):

self.queue = []

# Add an element

def enqueue(self, item):

self.queue.append(item)

# Remove an element

def dequeue(self):

if len(self.queue) < 1:

return None

return self.queue.pop(0)

# Display the queue

def display(self):

print(self.queue)

def size(self):

return len(self.queue)

q = Queue()

q.enqueue(1)

q.enqueue(2)

q.enqueue(3)

q.enqueue(4)

q.enqueue(5)

q.display()

q.dequeue()

print("After removing an element")

q.display()

**Practical 8: Infix to prefix**

def isOperator(c):

    return (not (c >= 'a' and c <= 'z') and not(c >= '0' and c <= '9') and not(c >= 'A' and c <= 'Z'))

def getPriority(C):

    if (C == '-' or C == '+'):

        return 1

    elif (C == '\*' or C == '/'):

        return 2

    elif (C == '^'):

        return 3

    return 0

def infixToPrefix(infix):

    operators = []

    operands = []

    for i in range(len(infix)):

        if (infix[i] == '('):

            operators.append(infix[i])

        elif (infix[i] == ')'):

            while (len(operators)!=0 and operators[-1] != '('):

                # operand 1

                op1 = operands[-1]

                operands.pop()

                # operand 2

                op2 = operands[-1]

                operands.pop()

                # operator

                op = operators[-1]

                operators.pop()

                tmp = op + op2 + op1

                operands.append(tmp)

            operators.pop()

        elif (not isOperator(infix[i])):

            operands.append(infix[i] + "")

        else:

            while (len(operators)!=0 and getPriority(infix[i]) <= getPriority(operators[-1])):

                op1 = operands[-1]

                operands.pop()

                op2 = operands[-1]

                operands.pop()

                op = operators[-1]

                operators.pop()

                tmp = op + op2 + op1

                operands.append(tmp)

            operators.append(infix[i])

    while (len(operators)!=0):

        op1 = operands[-1]

        operands.pop()

        op2 = operands[-1]

        operands.pop()

        op = operators[-1]

        operators.pop()

        tmp = op + op2 + op1

        operands.append(tmp)

    return operands[-1]

s = "(A-B/C)\*(A/K-L)"

print( infixToPrefix(s))

**Practical 9: Infix to postfix**

OPERATORS = set(['+', '-', '\*', '/', '(', ')', '^'])  # set of operators

PRIORITY = {'+':1, '-':1, '\*':2, '/':2, '^':3} # dictionary having priorities

def infix\_to\_postfix(expression): #input expression

    stack = [] # initially stack empty

    output = '' # initially output empty

    for ch in expression:

        if ch not in OPERATORS:  # if an operand then put it directly in postfix expression

            output+= ch

        elif ch=='(':  # else operators should be put in stack

            stack.append('(')

        elif ch==')':

            while stack and stack[-1]!= '(':

                output+=stack.pop()

            stack.pop()

        else:

            # lesser priority can't be on top on higher or equal priority

             # so pop and put in output

            while stack and stack[-1]!='(' and PRIORITY[ch]<=PRIORITY[stack[-1]]:

                output+=stack.pop()

            stack.append(ch)

    while stack:

        output+=stack.pop()

    return output

expression = input('Enter infix expression')

print('infix expression: ',expression)

print('postfix expression: ',infix\_to\_postfix(expression))

**Practical 10:**

**# Bubble sort in Python**

def bubbleSort(array):

  # loop to access each array element

  for i in range(len(array)):

    # loop to compare array elements

    for j in range(0, len(array) - i - 1):

      # compare two adjacent elements

      # change > to < to sort in descending order

      if array[j] > array[j + 1]:

        # swapping elements if elements

        # are not in the intended order

        temp = array[j]

        array[j] = array[j+1]

        array[j+1] = temp

data = [-2, 45, 0, 11, -9]

bubbleSort(data)

print('Sorted Array in Ascending Order:')

print(data)

**Practical 11:Insertion sort**

**# Insertion sort in Python**

def insertionSort(array):

#range starts from 1 since position 0 is considered fixed in 1st iteration

    for step in range(1, len(array)):

        temp = array[step]

        j = step - 1

        # Compare temp with each element on the left of it until an element smaller than it is found

        # For descending order, change temp<array[j] to temp>array[j].

        while j >= 0 and temp < array[j]:

            array[j + 1] = array[j]

            j = j - 1

        # Place temp at after the element just smaller than it.

        array[j + 1] = temp

data = [9, 5, 1, 4, 3]

insertionSort(data)

print('Sorted Array in Ascending Order:')

print(data)

**Practical 12:**

**# MergeSort in Python**

def mergeSort(array):

    if len(array) > 1:

        #  r is the point where the array is divided into two subarrays

        r = len(array)//2

        L = array[:r]

        M = array[r:]

        # Sort the two halves

        mergeSort(L)

        mergeSort(M)

        i = j = k = 0

        # Until we reach either end of either L or M, pick larger among

        # elements L and M and place them in the correct position at A[p..r]

        while i < len(L) and j < len(M):

            if L[i] < M[j]:

                array[k] = L[i]

                i += 1

            else:

                array[k] = M[j]

                j += 1

            k += 1

        # When we run out of elements in either L or M,

        # pick up the remaining elements and put in A[p..r]

        while i < len(L):

            array[k] = L[i]

            i += 1

            k += 1

        while j < len(M):

            array[k] = M[j]

            j += 1

            k += 1

# Print the array

def printList(array):

    for i in range(len(array)):

        print(array[i], end=" ")

    print()

# Driver program

if \_\_name\_\_ == '\_\_main\_\_':

    array = [6, 5, 12, 10, 9, 1]

    mergeSort(array)

    print("Sorted array is: ")

    printList(array)

**Practical 12: Looping programs like ( done in class)**

**a) Display 1st 10 natural numbers using for/while loop**

**b) Display multiplication table using for/while loop.**

**c) sum of natural number using loop.**

**Practical 13: Sparse matrix**

L = [[0, 0, 0, 1, 1],

     [0, 0, 0, 0, 0],

     [1, 0, 0, 0, 0]]

j = 0

non\_zero\_num = 0

zero\_num = 0

L\_1 = []

while j < len(L):

    i=0

    while i < (len(L[j])):

        if L[j][i] == 0:

            zero\_num += 1

        else:

            non\_zero\_num += 1

        i += 1

    j += 1

if non\_zero\_num < zero\_num:

    print("Given Matrix is Sparse")

else:

    print("Given Matrix is Dense")