Practicals

> **Program 1 :** Write a program to implement Abstract Data Types(ADT)

Stack ADT:

```
# Stack implementation in python

# Creating a stack
def create_stack():
  stack = []
  return stack

# Creating an empty stack
def check_empty(stack):
  return len(stack) == 0

# Adding items into the stack
def push(stack, item):
  stack.append(item)
  print("pushed item: " + item)

# Removing an element from the stack
def pop(stack):
  if (check_empty(stack)):
    return "stack is empty"

return stack.pop()
```

```
stack = create_stack()
push(stack, str(1))
push(stack, str(2))
push(stack, str(3))
push(stack, str(4))
print("popped item: " + pop(stack))
print("stack after popping an element: " + str(stack))
```

Output

```
pushed item: 3
pushed item: 4
popped item: 4
stack after popping an element: ['1', '2', '3']
```

➤ **Program 2 : Write a Program to implement Singly linked list with insertion, deletion, traversal operations**

```
# Linked list operations in Python


# Create a node
class Node:
  def __init__(self, data):
    self.data = data
    self.next = None


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


  # Insert at the beginning
  def insertAtBeginning(self, new_data):
    new_node = Node(new_data)
```

3 is found

Sorted List:

2 3 4 5

➢ **Program 3 : Write a Program to implement Doubly linked list with insertion, deletion, traversal operations**

```
import gc
# Initialise the Node
class Node:
  def __init__(self, data):
    self.item = data
    self.next = None
    self.prev = None
# Class for doubly Linked List
class doublyLinkedList:
  def __init__(self):
    self.start_node = None
  # Insert Element to Empty list
  def InsertToEmptyList(self, data):
    if self.start_node is None:
      new_node = Node(data)
      self.start_node = new_node
    else:
      print("The list is empty")
  # Insert element at the end
  def InsertToEnd(self, data):
    # Check if the list is empty
    if self.start_node is None:
      new_node = Node(data)
      self.start_node = new_node
      return
    n = self.start_node
```

```
    # Iterate till the next reaches NULL
    while n.next is not None:
        n = n.next
    new_node = Node(data)
    n.next = new_node
    new_node.prev = n
# Delete the elements from the start
def DeleteAtStart(self):
    if self.start_node is None:
        print("The Linked list is empty, no element to delete")
        return
    if self.start_node.next is None:
        self.start_node = None
        return
    self.start_node = self.start_node.next
    self.start_prev = None;
# Delete the elements from the end
def delete_at_end(self):
    # Check if the List is empty
    if self.start_node is None:
        print("The Linked list is empty, no element to delete")
        return
    if self.start_node.next is None:
        self.start_node = None
        return
    n = self.start_node
    while n.next is not None:
        n = n.next
    n.prev.next = None
# Traversing and Displaying each element of the list
def Display(self):
```

```
        if self.start_node is None:
            print("The list is empty")
            return
        else:
            n = self.start_node
            while n is not None:
                print("Element is: ", n.item)
                n = n.next
        print("\n")
# Create a new Doubly Linked List
NewDoublyLinkedList = doublyLinkedList()
# Insert the element to empty list
NewDoublyLinkedList.InsertToEmptyList(10)
# Insert the element at the end
NewDoublyLinkedList.InsertToEnd(20)
NewDoublyLinkedList.InsertToEnd(30)
NewDoublyLinkedList.InsertToEnd(40)
NewDoublyLinkedList.InsertToEnd(50)
NewDoublyLinkedList.InsertToEnd(60)
# Display Data
NewDoublyLinkedList.Display()
# Delete elements from start
NewDoublyLinkedList.DeleteAtStart()
# Delete elements from end
NewDoublyLinkedList.DeleteAtStart()
# Display Data
NewDoublyLinkedList.Display()
```

Output

Element is: 10

Element is: 20

Element is: 30

Element is: 40

Element is: 50

Element is: 60

➢ **Program 4 : Write a program to implement stack with insertion, deletion, traversal operations.**

```
# Stack implementation in python

# Creating a stack
def create_stack():
    stack = []
    return stack

# Creating an empty stack
def check_empty(stack):
    return len(stack) == 0

# Adding items into the stack
def push(stack, item):
    stack.append(item)
    print("pushed item: " + item)

# Removing an element from the stack
def pop(stack):
    if (check_empty(stack)):
        return "stack is empty"
```

```
    return stack.pop()

stack = create_stack()
push(stack, str(1))
push(stack, str(2))
push(stack, str(3))
push(stack, str(4))
print("popped item: " + pop(stack))
print("stack after popping an element: " + str(stack))
```

Output

```
pushed item: 1
pushed item: 2
pushed item: 3
pushed item: 4
popped item: 4
stack after popping an element: ['1', '2', '3']
```

➢ **Program 5 : Write a program to implement Queue with insertion, deletion, traversal operations.**

```
# Queue implementation in Python

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()
```

Output

```
[1, 2, 3, 4, 5]
After removing an element
[2, 3, 4, 5]
```

➢ **Program 6 : Write a program to implement Priority Queue with insertion, deletion, traversal operations.**

```
# Priority Queue implementation in Python

# Function to heapify the tree
def heapify(arr, n, i):
  # Find the largest among root, left child and right child
  largest = i
  l = 2 * i + 1
  r = 2 * i + 2

  if l < n and arr[i] < arr[l]:
    largest = l

  if r < n and arr[largest] < arr[r]:
    largest = r

  # Swap and continue heapifying if root is not largest
  if largest != i:
    arr[i], arr[largest] = arr[largest], arr[i]
    heapify(arr, n, largest)

# Function to insert an element into the tree
def insert(array, newNum):
  size = len(array)
  if size == 0:
    array.append(newNum)
  else:
    array.append(newNum)
    for i in range((size // 2) - 1, -1, -1):
      heapify(array, size, i)
```

```
# Function to delete an element from the tree
def deleteNode(array, num):
    size = len(array)
    i = 0
    for i in range(0, size):
        if num == array[i]:
            break

    array[i], array[size - 1] = array[size - 1], array[i]

    array.remove(size - 1)

    for i in range((len(array) // 2) - 1, -1, -1):
        heapify(array, len(array), i)

arr = []

insert(arr, 3)
insert(arr, 4)
insert(arr, 9)
insert(arr, 5)
insert(arr, 2)

print ("Max-Heap array: " + str(arr))
deleteNode(arr, 4)
print("After deleting an element: " + str(arr))
```

Output

```
Max-Heap array: [9, 5, 4, 3, 2]
After deleting an element: [9, 5, 2, 3]
```

> **Program 7 :** **Write a program to implement Binary tree with insertion, deletion, traversal operations .**

```
# Binary Tree in Python

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

  # Traverse preorder
  def traversePreOrder(self):
    print(self.val, end=' ')
    if self.left:
      self.left.traversePreOrder()
    if self.right:
      self.right.traversePreOrder()

  # Traverse inorder
  def traverseInOrder(self):
    if self.left:
      self.left.traverseInOrder()
    print(self.val, end=' ')
    if self.right:
      self.right.traverseInOrder()

  # Traverse postorder
  def traversePostOrder(self):
    if self.left:
      self.left.traversePostOrder()
    if self.right:
```

```
        self.right.traversePostOrder()
    print(self.val, end=' ')


root = Node(1)

root.left = Node(2)
root.right = Node(3)

root.left.left = Node(4)

print("Pre order Traversal: ", end="")
root.traversePreOrder()
print("\nIn order Traversal: ", end="")
root.traverseInOrder()
print("\nPost order Traversal: ", end="")
root.traversePostOrder()
```

Output

```
Pre order Traversal: 1 2 4 3
In order Traversal: 4 2 1 3
Post order Traversal: 4 2 3 1
```

➤ **Program 8 : Write a program to implement Huffman coding**

```
# Huffman Coding in python
string = 'BCAADDDCCACACAC'
# Creating tree nodes
class NodeTree(object):

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

```
    def children(self):
        return (self.left, self.right)

    def nodes(self):
        return (self.left, self.right)

    def __str__(self):
        return '%s_%s' % (self.left, self.right)
# Main function implementing huffman coding
def huffman_code_tree(node, left=True, binString=''):
    if type(node) is str:
        return {node: binString}
    (l, r) = node.children()
    d = dict()
    d.update(huffman_code_tree(l, True, binString + '0'))
    d.update(huffman_code_tree(r, False, binString + '1'))
    return d

# Calculating frequency
freq = {}
for c in string:
    if c in freq:
        freq[c] += 1
    else:
        freq[c] = 1

freq = sorted(freq.items(), key=lambda x: x[1], reverse=True)

nodes = freq

while len(nodes) > 1:
    (key1, c1) = nodes[-1]
```

```
    (key2, c2) = nodes[-2]
    nodes = nodes[:-2]
    node = NodeTree(key1, key2)
    nodes.append((node, c1 + c2))

    nodes = sorted(nodes, key=lambda x: x[1], reverse=True)

huffmanCode = huffman_code_tree(nodes[0][0])
print(' Char | Huffman code ')
print('----------------------')
for (char, frequency) in freq:
    print(' %-4r |%12s' % (char, huffmanCode[char]))
```

Output

```
Char | Huffman code
----------------------
'C' |      0
'A' |     11
'D' |    101
'B' |    100
```

➢ **Program 9 : Write a program to implement Graph with insertion, deletion, traversal operations**

```
# Python program for
# validation of a graph

# import dictionary for graph
from collections import defaultdict

# function for adding edge to graph
graph = defaultdict(list)
def addEdge(graph,u,v):
```

```
    graph[u].append(v)

# definition of function
def generate_edges(graph):
    edges = []

    # for each node in graph
    for node in graph:

        # for each neighbour node of a single node
        for neighbour in graph[node]:

            # if edge exists then append
            edges.append((node, neighbour))
    return edges

# declaration of graph as dictionary
addEdge(graph,'a','c')
addEdge(graph,'b','c')
addEdge(graph,'b','e')
addEdge(graph,'c','d')
addEdge(graph,'c','e')
addEdge(graph,'c','a')
addEdge(graph,'c','b')
addEdge(graph,'e','b')
addEdge(graph,'d','c')
addEdge(graph,'e','c')

# Driver Function call
# to print generated graph
print(generate_edges(graph))
```

Output

[('a', 'c'), ('b', 'c'), ('b', 'e'), ('c', 'd'), ('c', 'e'), ('c', 'a'), ('c', 'b'), ('e', 'b'), ('e', 'c'), ('d', 'c')]

> ## Program 10 : Write a program to implement Travelling Salesman Problem

```
# Python program to implement traveling salesman problem using naive
approach.
from sys import maxsize
from itertools import permutations
V = 4

# implementation of traveling Salesman Problem
def travellingSalesmanProblem(graph, s):

    # store all vertex apart from source vertex
    vertex = []
    for i in range(V):
        if i != s:
            vertex.append(i)

    # store minimum weight Hamiltonian Cycle
    min_path = maxsize
    next_permutation=permutations(vertex)
    for i in next_permutation:

        # store current Path weight(cost)
        current_pathweight = 0

        # compute current path weight
        k = s
        for j in i:
```

```
            current_pathweight += graph[k][j]
            k = j
        current_pathweight += graph[k][s]

        # update minimum
        min_path = min(min_path, current_pathweight)

    return min_path

# Driver Code
if __name__ == "__main__":

    # matrix representation of graph
    graph = [[0, 10, 15, 20], [10, 0, 35, 25],
             [15, 35, 0, 30], [20, 25, 30, 0]]
    s = 0
    print(travellingSalesmanProblem(graph, s)) # Python3 program to
implement traveling salesman
# problem using naive approach.
from sys import maxsize
from itertools import permutations
V = 4

# implementation of traveling Salesman Problem
def travellingSalesmanProblem(graph, s):

    # store all vertex apart from source vertex
    vertex = []
    for i in range(V):
        if i != s:
            vertex.append(i)
```

```
    # store minimum weight Hamiltonian Cycle
    min_path = maxsize
    next_permutation=permutations(vertex)
    for i in next_permutation:

        # store current Path weight(cost)
        current_pathweight = 0

        # compute current path weight
        k = s
        for j in i:
            current_pathweight += graph[k][j]
            k = j
        current_pathweight += graph[k][s]

        # update minimum
        min_path = min(min_path, current_pathweight)

    return min_path

# Driver Code
if __name__ == "__main__":

    # matrix representation of graph
    graph = [[0, 10, 15, 20], [10, 0, 35, 25],
             [15, 35, 0, 30], [20, 25, 30, 0]]
    s = 0
    print(travellingSalesmanProblem(graph, s))
```

Output

```
80
```

> **Program 11 : Write a program to create basic hash table for insertion, deletion, traversal operations(assume that there is no collisions)**

```
# Python program to demonstrate working of HashTable

hashTable = [[],] * 10

def checkPrime(n):
    if n == 1 or n == 0:
        return 0

    for i in range(2, n//2):
        if n % i == 0:
            return 0

    return 1

def getPrime(n):
    if n % 2 == 0:
        n = n + 1

    while not checkPrime(n):
        n += 2

    return n

def hashFunction(key):
    capacity = getPrime(10)
    return key % capacity

def insertData(key, data):
```

```
    index = hashFunction(key)
    hashTable[index] = [key, data]

def removeData(key):
    index = hashFunction(key)
    hashTable[index] = 0

insertData(123, "C")
insertData(432, "Python")
insertData(213, "JAVA")
insertData(654, "C++")

print(hashTable)

removeData(123)

print(hashTable)
```

Output

```
[[], [], [123, 'C'], [432, 'Python'], [213, 'JAVA'], [654, 'C++'], [], [], [], []]
[[], [], 0, [432, 'Python'], [213, 'JAVA'], [654, 'C++'], [], [], [], []]
```

➢ **Program 12 : Write a program to create hash table to handle collisions using overflow chaining.**

```
# Function to display hashtable
def display_hash(hashTable):
    for i in range(len(hashTable)):
        print(i, end=" ")

        for j in hashTable[i]:
            print("-->", end=" ")
            print(j, end=" ")
```

```
        print()

# Creating Hashtable as
# a nested list.
HashTable = [[] for _ in range(10)]

# Hashing Function to return
# key for every value.
def Hashing(keyvalue):
    return keyvalue % len(HashTable)

# Insert Function to add
# values to the hash table
def insert(Hashtable, keyvalue, value):
    hash_key = Hashing(keyvalue)
    Hashtable[hash_key].append(value)

# Driver Code
insert(HashTable, 11, 'JAVA')
insert(HashTable, 3, 'PYTHON')
insert(HashTable, 10, 'C')
insert(HashTable, 9, 'C++')
insert(HashTable, 21, 'JAVASCRIPT')
insert(HashTable, 20, 'HTML')
insert(HashTable, 4, 'PHP')

display_hash(HashTable)
```

Output

```
0 --> C --> HTML
1 --> JAVA --> JAVASCRIPT
2
```

3 --> PYTHON

4 --> PHP

5

6

7

8

9 --> C++