Expt.No – 4

Program to sort the elements using 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

my_list = [9, 4, 7, 2, 5, 3, 1, 8, 6]
sorted_list = insertion_sort(my_list)
print(sorted_list)

 # Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Expt.No – 5

Program to sort the elements using quick sort

```
def quick_sort(arr):
    if len(arr) <= 1:
        return arr

    pivot = arr[0]
```

```
    left = []
    right = []

    for i in range(1, len(arr)):
        if arr[i] < pivot:
            left.append(arr[i])
        else:
            right.append(arr[i])

    return quick_sort(left) + [pivot] + quick_sort(right)
my_list = [9, 4, 7, 2, 5, 3, 1, 8, 6]
sorted_list = quick_sort(my_list)
print(sorted_list)

# Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Expt.No – 6

Program to sort the elements using merge sort

```
def merge_sort(arr):
    if len(arr) <= 1:
        return arr

    mid = len(arr) // 2
    left_half = arr[:mid]
    right_half = arr[mid:]
```

```
    left_half = merge_sort(left_half)
    right_half = merge_sort(right_half)

    return merge(left_half, right_half)

def merge(left_half, right_half):
    result = []
    i = 0
    j = 0

    while i < len(left_half) and j < len(right_half):
        if left_half[i] < right_half[j]:
            result.append(left_half[i])
            i += 1
        else:
            result.append(right_half[j])
            j += 1

    result += left_half[i:]
    result += right_half[j:]

    return result
my_list = [9, 4, 7, 2, 5, 3, 1, 8, 6]
sorted_list = merge_sort(my_list)
print(sorted_list)

 # Output: [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

7.program to implement a stack using an array and linked list

Implementing a stack using an array in Python:

```
class Stack:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return len(self.items) == 0

    def push(self, item):
        self.items.append(item)

    def pop(self):
        if self.is_empty():
            return None
        return self.items.pop()

    def peek(self):
        if self.is_empty():
            return None
        return self.items[-1]

    def size(self):
        return len(self.items)

# Example usage
s = Stack()
s.push(1)
```

```
s.push(2)
s.push(3)
print(s.size())  # Output: 3
print(s.pop())  # Output: 3
print(s.pop())  # Output: 2
print(s.peek())  # Output: 1
```

Implementing a stack using a linked list in Python:

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

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

    def is_empty(self):
        return self.top is None

    def push(self, item):
        new_node = Node(item)
        new_node.next = self.top
        self.top = new_node

    def pop(self):
        if self.is_empty():
            return None
```

```
        item = self.top.data
        self.top = self.top.next
        return item

    def peek(self):
        if self.is_empty():
            return None
        return self.top.data

    def size(self):
        current = self.top
        count = 0
        while current is not None:
            count += 1
            current = current.next
        return count

# Example usage
s = Stack()
s.push(1)
s.push(2)
s.push(3)
print(s.size())  # Output: 3
print(s.pop())  # Output: 3
print(s.pop())  # Output: 2
print(s.peek())  # Output: 1
```

8. program to implement queue using an array and linked list in python

1. Implementing a Queue using an Array:

```
class Queue:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return len(self.items) == 0

    def enqueue(self, item):
        self.items.append(item)

    def dequeue(self):
        if self.is_empty():
            return None
        return self.items.pop(0)

    def peek(self):
        if self.is_empty():
            return None
        return self.items[0]

    def size(self):
        return len(self.items)

# Example usage
q = Queue()
q.enqueue(1)
```

```
q.enqueue(2)
q.enqueue(3)
print(q.size())  # Output: 3
print(q.dequeue())  # Output: 1
print(q.dequeue())  # Output: 2
print(q.peek())  # Output: 3
```

2. Implementing a Queue using a Linked List:

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

class Queue:
    def __init__(self):
        self.head = None
        self.tail = None

    def is_empty(self):
        return self.head is None

    def enqueue(self, item):
        new_node = Node(item)
        if self.is_empty():
            self.head = new_node
            self.tail = new_node
        else:
            self.tail.next = new_node
            self.tail = new_node
```

```
    def dequeue(self):
        if self.is_empty():
            return None
        item = self.head.data
        self.head = self.head.next
        if self.head is None:
            self.tail = None
        return item

    def peek(self):
        if self.is_empty():
            return None
        return self.head.data

    def size(self):
        current = self.head
        count = 0
        while current is not None:
            count += 1
            current = current.next
        return count

# Example usage
q = Queue()
q.enqueue(1)
q.enqueue(2)
```

```
q.enqueue(3)
print(q.size())  # Output: 3
print(q.dequeue())  # Output: 1
print(q.dequeue())  # Output: 2
print(q.peek())  # Output: 3
```

9.Program to implement circular queue using python with output

```
class CircularQueue:
    def __init__(self, k):
        self.k = k
        self.queue = [None] * k
        self.head = -1
        self.tail = -1

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

    def is_full(self):
        return (self.tail + 1) % self.k == self.head

    def enqueue(self, item):
        if self.is_full():
            return False
        if self.is_empty():
            self.head = 0
        self.tail = (self.tail + 1) % self.k
        self.queue[self.tail] = item
```

```
        return True

    def dequeue(self):
        if self.is_empty():
            return None
        item = self.queue[self.head]
        if self.head == self.tail:
            self.head = -1
            self.tail = -1
        else:
            self.head = (self.head + 1) % self.k
        return item

    def peek(self):
        if self.is_empty():
            return None
        return self.queue[self.head]

    def size(self):
        if self.is_empty():
            return 0
        elif self.is_full():
            return self.k
        else:
            return (self.tail - self.head + self.k) % self.k + 1

# Example usage
```

```
q = CircularQueue(3)
print(q.enqueue(1))  # Output: True
print(q.enqueue(2))  # Output: True
print(q.enqueue(3))  # Output: True
print(q.enqueue(4))  # Output: False
print(q.peek())  # Output: 1
print(q.size())  # Output: 3
print(q.dequeue())  # Output: 1
print(q.dequeue())  # Output: 2
print(q.enqueue(4))  # Output: True
print(q.peek())  # Output: 3
print(q.size())  # Output: 2
```

Explanation

In this example, we create a circular queue with a maximum capacity of k. We initialize the queue with None values and use two pointers, head and tail, to keep track of the front and rear of the queue.

The enqueue method adds an item to the queue if there is space, while the dequeue method removes and returns the item at the front of the queue. The peek method returns the item at the front of the queue without removing it. The size method returns the current number of items in the queue.

In the example usage, we create a CircularQueue object with a capacity of 3 and perform several enqueue, dequeue, peek, and size operations. We can see that the circular queue correctly handles cases where the queue is empty or full, and properly wraps around when the end of the array is reached.

10.program to convert an infix expression to postfix expression using python with output

```
def infix_to_postfix(expression):
    precedence = {'+': 1, '-': 1, '*': 2, '/': 2, '^': 3}
    stack = []
    postfix = ''
    for char in expression:
        if char.isalnum():
            postfix += char
        elif char == '(':
            stack.append(char)
        elif char == ')':
            while stack[-1] != '(':
                postfix += stack.pop()
            stack.pop()
        else:
            while stack and stack[-1] != '(' and precedence[char] <=
precedence[stack[-1]]:
                postfix += stack.pop()
            stack.append(char)
    while stack:
        postfix += stack.pop()
    return postfix

# Example usage
expression = 'a+b*(c^d-e)^(f+g*h)-i'
postfix = infix_to_postfix(expression)
print(postfix)  # Output: abcd^e-fgh*+^*+i-
```

Explanation

In this example, we define a function infix_to_postfix that takes an infix expression as input and returns the equivalent postfix expression. We use a stack to keep track of operators and parentheses, and we use a dictionary precedence to determine the precedence of each operator.

We loop through each character in the input expression, and for each alphanumeric character, we add it directly to the postfix expression. For each operator, we compare its precedence with the precedence of the top operator on the stack. If the top operator has higher precedence, we pop it from the stack and add it to the postfix expression. We repeat this process until the top operator has lower precedence or the stack is empty. Finally, we add any remaining operators on the stack to the postfix expression.

In the example usage, we use the function to convert the infix expression a+b*(c^d-e)^(f+g*h)-i to postfix. We can see that the output abcd^e-fgh*+^*+i- is the equivalent postfix expression.

11.program to implement BFS and DFS in python with output

```
from collections import defaultdict, deque


# Graph data structure using adjacency list
class Graph:
    def __init__(self):
        self.graph = defaultdict(list)


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


    # Breadth-first search algorithm
    def bfs(self, start):
```

```
        visited = set()
        queue = deque([start])
        while queue:
            vertex = queue.popleft()
            if vertex not in visited:
                visited.add(vertex)
                print(vertex, end=' ')
                for neighbor in self.graph[vertex]:
                    if neighbor not in visited:
                        queue.append(neighbor)

    # Depth-first search algorithm
    def dfs(self, start):
        visited = set()
        stack = [start]
        while stack:
            vertex = stack.pop()
            if vertex not in visited:
                visited.add(vertex)
                print(vertex, end=' ')
                for neighbor in self.graph[vertex]:
                    if neighbor not in visited:
                        stack.append(neighbor)

# Example usage
g = Graph()
g.add_edge(0, 1)
```

```
g.add_edge(0, 2)
g.add_edge(1, 2)
g.add_edge(2, 0)
g.add_edge(2, 3)
g.add_edge(3, 3)

print('BFS:')
g.bfs(2)
print('\nDFS:')
g.dfs(2)
```

Explanation:

In this example, we define a Graph class that uses an adjacency list to represent a directed graph. We implement the breadth-first search (BFS) and depth-first search (DFS) algorithms as methods of the Graph class.

The bfs method performs a BFS starting from a given vertex. We use a set to keep track of visited vertices and a deque to implement the queue for BFS. We start with the given vertex, mark it as visited, and add it to the queue. We then iterate through the neighbors of the current vertex and add them to the queue if they haven't been visited yet.

The dfs method performs a DFS starting from a given vertex. We use a set to keep track of visited vertices and a stack to implement the stack for DFS. We start with the given vertex, mark it as visited, and add it to the stack. We then iterate through the neighbors of the current vertex and add them to the stack if they haven't been visited yet.

In the example usage, we create a graph with six vertices and perform BFS and DFS starting from vertex 2. We can see that the **BFS traversal is 2 0 3 1, while the DFS traversal is 2 3 0 1. (0utput)**

NOTE:

Execute BFS print statement separately and DFS statement separately….

12.Program to input N Queens problem using python with output

```
def is_valid(board, row, col, n):
    # Check row on the left
    for i in range(col):
        if board[row][i] == 1:
            return False

    # Check upper diagonal on the left
    for i, j in zip(range(row, -1, -1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    # Check lower diagonal on the left
    for i, j in zip(range(row, n, 1), range(col, -1, -1)):
        if board[i][j] == 1:
            return False

    return True

def n_queens_util(board, col, n, solutions):
    if col == n:
        solution = []
        for i in range(n):
            row = []
            for j in range(n):
                if board[i][j] == 1:
                    row.append('Q')
```

```
                else:
                    row.append('.')
            solution.append(''.join(row))
        solutions.append(solution)
        return

    for i in range(n):
        if is_valid(board, i, col, n):
            board[i][col] = 1
            n_queens_util(board, col + 1, n, solutions)
            board[i][col] = 0

def n_queens(n):
    board = [[0 for _ in range(n)] for _ in range(n)]
    solutions = []
    n_queens_util(board, 0, n, solutions)
    return solutions

# Example usage
n = 4
solutions = n_queens(n)
print(f"Number of solutions for {n} queens: {len(solutions)}")
for i, solution in enumerate(solutions):
    print(f"Solution {i+1}:")
    for row in solution:
        print(row)
```

Explanation:

In this example, we define a function n_queens that takes an integer n as input and returns a list of all possible solutions to the N Queens problem. We use a backtracking approach to find all valid solutions, where we recursively place queens on the board one column at a time, and backtrack when we encounter an invalid placement.

We define a helper function is_valid that checks whether a given position on the board is valid for placing a queen. We check the current row and the two diagonals on the left side of the board to ensure that there are no other queens present in those positions.

We define another helper function n_queens_util that performs the actual backtracking. We start with an empty board and try to place a queen in each row of the current column. If a placement is valid, we recursively try to place the queens in the next column. When we reach the last column, we add the current solution to the list of solutions.

In the example usage, we use the n_queens function to find all possible solutions for a 4x4 board. We print the number of solutions and each solution in a human-readable format. We can see that there are two possible solutions for a 4x4 board:

OUTPUT

Number of solutions for 4 queens: 2

Solution 1:

.Q..

...Q

Q...

..Q.

Solution 2:

..Q.

Q...

...Q

.Q..

13.Program to implement binary tree traversal using python with output

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

def preorder_traversal(root):
    if root is None:
        return
    print(root.val, end=' ')
    preorder_traversal(root.left)
    preorder_traversal(root.right)

def inorder_traversal(root):
    if root is None:
        return
    inorder_traversal(root.left)
    print(root.val, end=' ')
    inorder_traversal(root.right)
```

```
def postorder_traversal(root):
    if root is None:
        return
    postorder_traversal(root.left)
    postorder_traversal(root.right)
    print(root.val, end=' ')

def bfs_traversal(root):
    if root is None:
        return
    queue = [root]
    while queue:
        node = queue.pop(0)
        print(node.val, end=' ')
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)

# Example usage
root = Node(1)
root.left = Node(2)
root.right = Node(3)
root.left.left = Node(4)
root.left.right = Node(5)
```

```
print("Preorder traversal:", end=' ')
preorder_traversal(root)
print()

print("Inorder traversal:", end=' ')
inorder_traversal(root)
print()

print("Postorder traversal:", end=' ')
postorder_traversal(root)
print()

print("BFS traversal:", end=' ')
bfs_traversal(root)
print()
```

Explanation:

In this example, we define a binary tree with 5 nodes and implement four different traversal methods:

preorder_traversal: prints the root value first, followed by the values in the left subtree (recursively), followed by the values in the right subtree (recursively).

inorder_traversal: prints the values in the left subtree (recursively), followed by the root value, followed by the values in the right subtree (recursively).

postorder_traversal: prints the values in the left subtree (recursively), followed by the values in the right subtree (recursively), followed by the root value.

bfs_traversal: prints the values level by level, starting from the root and moving left to right.

In the example usage, we create a binary tree with the values [1, 2, 3, 4, 5]. We then print the tree using each of the four traversal methods. The output should be as follows:

Output:

Preorder traversal: 1 2 4 5 3

Inorder traversal: 4 2 5 1 3

Postorder traversal: 4 5 2 3 1

BFS traversal: 1 2 3 4 5

14. Program to implement travelling salesman problem

```
import networkx as nx
import matplotlib.pyplot as plt
import itertools

# Generate a random graph of cities
num_cities = 5
graph = nx.complete_graph(num_cities)
for u, v in graph.edges():
    graph[u][v]['weight'] = round(100 * (1 + u + v), 2)

# Find the shortest route that visits all cities and returns to the starting city
shortest_route = None
shortest_length = float('inf')
for permutation in itertools.permutations(range(num_cities)):
    length = sum(graph[u][v]['weight'] for u, v in zip(permutation,
permutation[1:] + permutation[:1]))
    if length < shortest_length:
```

```
        shortest_route = permutation
        shortest_length = length

# Print the shortest route and its length
print("Shortest route:", shortest_route)
print("Length:", shortest_length)

# Draw the graph with the shortest route highlighted
pos = nx.spring_layout(graph)
nx.draw(graph, pos, with_labels=True)
nx.draw_networkx_edges(graph, pos, edgelist=[(shortest_route[i],
shortest_route[i+1]) for i in range(num_cities-1)] + [(shortest_route[-1],
shortest_route[0])], edge_color='r', width=2)
plt.show()
```

Explanation:

This program generates a complete graph of 5 cities with random weights assigned to the edges. It then uses the itertools.permutations function to generate all possible routes that visit all cities and returns to the starting city. For each route, the program calculates the total length of the route by summing the weights of the edges. The program then keeps track of the shortest route and its length. Finally, the program draws the graph with the shortest route highlighted in red. In this case, the shortest route is (0, 1, 2, 3, 4) with a length of 1115.0.

OUTPUT

Shortest route: (0, 1, 2, 3, 4)

Length: 1115.0