PRACTICAL NO: 12

 # Define maximum size MAX

= 20

```
# Define Employee class
class Employee:
    def init (self, name="", code=0, designation="", exp=0, age=0, salary=0):
        self.name = name
        self.code = code
        self.designation =
        designation self.exp = exp
        self.age = age
        self.salary = salary


# Global variable
num = 0
emp = [Employee() for _ in range(MAX)]   # List of Employee objects


# Function to show menu
def show_menu():
```

```
running = True
while running:
    print("University Employee Management System")
    print("Available Options:")
    print("Build Table (1)")
    print("Insert New Entry (2)")
    print("Delete Entry (3)")
    print("Search a Record (4)")
    print("Exit (5)")

    # Input Options
    try:
        option = int(input())
    except ValueError:
        print("Invalid input. Please enter a number between 1 and 5.")
        continue

    # Call function based on the
    option if option == 1:
        build()
    elif option == 2:
        insert()
    elif option == 3:
        delete_record()
    elif option == 4:
        search_record()
```

```
        elif option == 5:
            running = False # Exit the loop and end the program
        else:
            print("Expected Options are 1/2/3/4/5")



# Function to build the given datatype
def build():
    global num
    print("Build The Table")
    print("Maximum Entries can be:",
    MAX)
    num = int(input("Enter the number of Entries required:
    ")) if num > 20:
        print("Maximum number of Entries are 20")
        num = 20

    print("Enter the following
    data:") for i in range(num):
        emp[i].name = input(f"Name of Employee {i + 1}: ")
        emp[i].code = int(input(f"Employee ID of Employee {i +
        1}: ")) emp[i].designation = input(f"Designation of
        Employee {i + 1}: ") emp[i].exp = int(input(f"Experience
        of Employee {i + 1}: ")) emp[i].age = int(input(f"Age of
        Employee {i + 1}: ")) emp[i].salary = int(input(f"Salary
        of Employee {i + 1}: "))
```

```
# Function to insert a new
employee def insert():
    global num
    if num < MAX:
        i = num
        num += 1
        print("Enter the information of the Employee")
        emp[i].name = input("Name: ")
        emp[i].code = int(input("Employee ID:
        ")) emp[i].designation =
        input("Designation: ") emp[i].exp =
        int(input("Experience: ")) emp[i].age =
        int(input("Age: ")) emp[i].salary =
        int(input("Salary: "))
    else:
        print("Employee Table Full")


# Function to delete an employee by
index def delete_index(i):
    global num
    for j in range(i, num - 1):
        emp[j] = emp[j + 1]
    num -= 1  # Decrease number of entries


# Function to delete a record
```

```
def delete_record():

    code = int(input("Enter the Employee ID to Delete Record:

    ")) for i in range(num):

        if emp[i].code == code:

            delete_index(i)

            break



# Function to search for a record by Employee

ID def search_record():

    code = int(input("Enter the Employee ID to Search

    Record: ")) for i in range(num):

        if emp[i].code == code:

            print(f"Name:

            {emp[i].name}")

            print(f"Employee ID: {emp[i].code}")

            print(f"Designation: {emp[i].designation}")

            print(f"Experience: {emp[i].exp}")

            print(f"Age: {emp[i].age}")

            print(f"Salary:

            {emp[i].salary}") break



# Driver Code

if    name__ == "    main    ":

    show_menu()  # Start the

    menu
```

OUTPUT---

```
University Employee Management System
Available Options:
Build Table (1)
Insert New Entry (2)
Delete Entry (3)
Search a Record (4)
Exit (5)
1
Build The Table
Maximum Entries can be: 20
Enter the number of Entries required: 2
Enter the following data:
Name of Employee 1: Shreyash
Employee ID of Employee 1: 123
Designation of Employee 1: Developer
Experience of Employee 1: 1
Age of Employee 1: 20
Salary of Employee 1: 60000
Name of Employee 2: Tanishkk
Employee ID of Employee 2: 456
Designation of Employee 2: manager
Experience of Employee 2: 5
Age of Employee 2: 30
Salary of Employee 2: 130000
University Employee Management System
Available Options:
Build Table (1)
Insert New Entry (2)
Delete Entry (3)
Search a Record (4)
Exit (5)
2
Enter the information of the Employee
Name: Varad
Employee ID: 678
Designation: developer
Experience: 2
Age: 22
Salary: 50000
University Employee Management System
Available Options:
```

```
Available Options:
Build Table (1)
Insert New Entry (2)
Delete Entry (3)
Search a Record (4)
Exit (5)
3
Enter the Employee ID to Delete Record: 678
University Employee Management System
Available Options:
Build Table (1)
Insert New Entry (2)
Delete Entry (3)
Search a Record (4)
Exit (5)
4
Enter the Employee ID to Search Record: 678
University Employee Management System
Available Options:
Build Table (1)
Insert New Entry (2)
Delete Entry (3)
Search a Record (4)
Exit (5)
4
Enter the Employee ID to Search Record: 123
Name: Shreyash
Employee ID: 123
Designation: Developer
Experience: 1
Age: 20
Salary: 60000
University Employee Management System
Available Options:
Build Table (1)
Insert New Entry (2)
Delete Entry (3)
Search a Record (4)
Exit (5)
5
```

PRACTICAL NO: 11

```
import pickle
import os

class Student:
    def __init__(self):
        self.roll_no = 0
        self.name = ""
        self.address = ""
        self.division = ""

    # Function to get student data
    def get_data(self):
        self.name = input("Enter your name: ")
        self.roll_no = int(input("Enter your Roll No.: "))
        self.division = input("Enter your Division: ")
        self.address = input("Enter your Address: ")

    # Function to show student data
    def show_data(self):
        print(f"\nName: {self.name}")
        print(f"Roll No.: {self.roll_no}")
        print(f"Division: {self.division}")
        print(f"Address: {self.address}")

    def get_roll(self):
        return self.roll_no

def add():
    with open("student.dat", "ab") as file:
        student = Student()
        student.get_data()
        pickle.dump(student, file)

def display():
    try:
        with open("student.dat", "rb") as file:
            while True:
                student = pickle.load(file)
                student.show_data()
    except EOFError:
        pass

def search(roll_no):
    found = False
    try:
        with open("student.dat", "rb") as file:
            while True:
                student = pickle.load(file)
                if student.get_roll() == roll_no:
                    student.show_data()
                    found = True
                    break
    except EOFError:
```

```
        if not found:
            print("Record not found!")

# Function to delete a student record by roll number
def delete(roll_no):
    found = False
    students = []

    try:
        with open("student.dat", "rb") as file:
            while True:
                student = pickle.load(file)
                if student.get_roll() != roll_no:
                    students.append(student)
                else:
                    found = True
    except EOFError:
        if found:
            with open("student.dat", "wb") as file:
                for student in students:
                    pickle.dump(student, file)
            print("\nRecord Deleted")
        else:
            print("Record not found!")

def main():
    while True:
        print("\nFile Handling")
        print("1) Add")
        print("2) Display")
        print("3) Search")
        print("4) Delete")
        print("5) Quit")

        choice = int(input("\nEnter your choice: "))

        if choice == 1:
            add()
        elif choice == 2:
            print("List of records:")
            display()
        elif choice == 3:
            roll_no = int(input("Enter Student Roll No: "))
            search(roll_no)
        elif choice == 4:
            roll_no = int(input("Enter Roll No to be deleted: "))
            delete(roll_no)
        elif choice == 5:
            print("End")
            break
        else:
            print("Invalid choice. Please try again.")

if __name__ == "__main__":
    main()
```

OUTPUT

```
File Handling
1) Add
2) Display
3) Search
4) Delete
5) Quit

Enter your choice: 1
Enter your name: Shreyash
Enter your Roll No.: 10
Enter your Division: A
Enter your Address: Sambhaji nagar

File Handling
1) Add
2) Display
3) Search
4) Delete
5) Quit

Enter your choice: 1
Enter your name: Tanishkk
Enter your Roll No.: 20
Enter your Division: B
Enter your Address: Pune

File Handling
1) Add
2) Display
3) Search
4) Delete
5) Quit

Enter your choice: 2
List of records:

Name: Shreyash
Roll No.: 41
Division: A
Address: Sambhaji nagar

Name: Tanishkk
Roll No.: 50
Division: B
Address: Pune

Name: Shreyash
Roll No.: 10
Division: A
Address: Sambhaji nagar
```

```
Name: Shreyash
Roll No.: 10
Division: A
Address: Sambhaji nagar

Name: Tanishkk
Roll No.: 20
Division: B
Address: Pune

File Handling
1) Add
2) Display
3) Search
4) Delete
5) Quit

Enter your choice: 3
Enter Student Roll No: 10

Name: Shreyash
Roll No.: 10
Division: A
Address: Sambhaji nagar

File Handling
1) Add
2) Display
3) Search
4) Delete
5) Quit

Enter your choice: 4
Enter Roll No to be deleted: 20

Record Deleted

File Handling
1) Add
2) Display
3) Search
4) Delete
5) Quit

Enter your choice: 5
End


...Program finished with exit code 0
Press ENTER to exit console.
```

PRACTICAL NO: 10

```
class Heap:

    def    init    (self):

        self.n = 0

        self.minheap =

        [] self.maxheap

        = []


    # Method to get input from the user and construct

    heaps def get(self):

        self.n = int(input("Enter number of students: "))


        print("Enter marks of students: ")

        for i in range(self.n):

            k = int(input()) # Input marks of

            student self.minheap.append(k)

            self.upAdjust(True, i) # Adjust for min

            heap self.maxheap.append(k)

            self.upAdjust(False, i)# Adjust for max heap


    # Method to display minimum mark (root of min

    heap) def displayMin(self):
```

```
        print(f"Minimum marks are: {self.minheap[0]}")

    # Method to display maximum mark (root of max
    heap)  def displayMax(self):
        print(f"Maximum marks are: {self.maxheap[0]}")

    # Method to adjust the heap (either min or max)
    def upAdjust(self, isMinHeap, i):
        if isMinHeap:  # Min Heap
            while i > 0 and self.minheap[(i - 1) // 2] > self.minheap[i]:
                self.minheap[i], self.minheap[(i - 1) // 2] = self.minheap[(i - 1) // 2],
                self.minheap[i]  i = (i - 1) // 2
        else:    # Max Heap
            while i > 0 and self.maxheap[(i - 1) // 2] < self.maxheap[i]:
                self.maxheap[i], self.maxheap[(i - 1) // 2] = self.maxheap[(i - 1) // 2],
                self.maxheap[i]  i = (i - 1) // 2

# Main function
if    name__ == "    main
    ":  H = Heap()
    H.get()
    H.displayMin()
    H.displayMax()
```

OUTPUT

```
Enter number of students: 5
Enter marks of students:
89
57
23
48
99
Minimum marks are: 23
Maximum marks are: 99


...Program finished with exit code 0
Press ENTER to exit console.
```

PRACTICAL NO: 9

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


class Dict:
    def __init__(self):
        self.root = None

    def asc(self, curr):
        if curr:
            self.asc(curr.left)
            print(f"{curr.word} - {curr.mean}")
            self.asc(curr.right)

    def desc(self, root):
        if root:
            self.desc(root.right)
            print(f"{root.word} - {root.mean}")
            self.desc(root.left)

    def inorder(self):
        self.asc(self.root)

    def postorder(self):
        self.desc(self.root)

    def insert(self, word, mean):
        node = Node(word, mean)
        if not self.root:
            self.root = node
            return True

        curr = self.root
        par = None
        while curr:
            if curr.word > word:
                par = curr
                curr = curr.left
            elif curr.word < word:
                par = curr
                curr = curr.right
            else:
                print("Same words not allowed")
                return False

        if par.word > word:
            par.left = node
        else:
            par.right = node
```

```
        return True

    def search(self, word, cnt):
        curr = self.root
        while curr:
            if curr.word > word:
                cnt += 1
                curr = curr.left
            elif curr.word < word:
                cnt += 1
                curr = curr.right
            elif curr.word == word:
                cnt += 1
                break
        return cnt

    def searchS(self, word):
        curr = self.root
        while curr:
            if curr.word > word:
                curr = curr.left
            elif curr.word < word:
                curr = curr.right
            elif curr.word == word:
                break
        return curr

    def update(self, word):
        nw = input(f"Enter new meaning of word {word} - ")
        curr = self.searchS(word)
        if curr:
            curr.mean = nw
        else:
            print("Word not found.")


def main():
    d = Dict()
    while True:
        print("\n--Dictionary--")
        print("1) Add word\n2) Display in Ascending\n3) Display in Descending\n4) Update\n5) Search\n6) 
Exit")
        try:
            ch = int(input("Enter your choice: "))
        except ValueError:
            print("Invalid input! Please enter a valid choice.")
            continue

        if ch == 1:
            w = input("Enter the word: ")
            m = input("Enter the meaning: ")
            d.insert(w, m)

        elif ch == 2:
```

```
            print("Ascending order is:")
            d.inorder()

        elif ch == 3:
            print("Descending order is:")
            d.postorder()

        elif ch == 4:
            uw = input("Enter word you want to update: ")
            d.update(uw)

        elif ch == 5:
            sw = input("Enter word you want to search: ")
            cnt = 0
            cnt = d.search(sw, cnt)
            if cnt == 0:
                print("Word not found")
            else:
                print(f"The comparisons required are: {cnt}")

        elif ch == 6:
            print("End")
            break

        else:
            print("Invalid choice. Please try again.")


if __name__ == "__main__":
    main()
```

OUTPUT

```
--Dictionary--
1) Add word
2) Display in Ascending
3) Display in Descending
4) Update
5) Search
6) Exit
Enter your choice: 1
Enter the word: A
Enter the meaning: apple

--Dictionary--
1) Add word
2) Display in Ascending
3) Display in Descending
4) Update
5) Search
6) Exit
Enter your choice: 1
Enter the word: B
Enter the meaning: bannana

--Dictionary--
1) Add word
2) Display in Ascending
3) Display in Descending
4) Update
5) Search
6) Exit
Enter your choice: 2
Ascending order is:
A - apple
B - bannana

--Dictionary--
1) Add word
2) Display in Ascending
3) Display in Descending
4) Update
5) Search
6) Exit
Enter your choice: 3
Descending order is:
B - bannana
A - apple

--Dictionary--
```

```
--Dictionary--
1) Add word
2) Display in Ascending
3) Display in Descending
4) Update
5) Search
6) Exit
Enter your choice: 4
Enter word you want to update: A
Enter new meaning of word A - application

--Dictionary--
1) Add word
2) Display in Ascending
3) Display in Descending
4) Update
5) Search
6) Exit
Enter your choice: 5
Enter word you want to search: A
The comparisons required are: 1

--Dictionary--
1) Add word
2) Display in Ascending
3) Display in Descending
4) Update
5) Search
6) Exit
Enter your choice: 5
Enter word you want to search: B
The comparisons required are: 2

--Dictionary--
1) Add word
2) Display in Ascending
3) Display in Descending
4) Update
5) Search
6) Exit
Enter your choice: 6
End


...Program finished with exit code 0
Press ENTER to exit console.
```

PRACTICAL NO: 8

```
import sys

class OBST:
    def __init__(self):
        self.SIZE = 10
        self.p = [0] * self.SIZE  # Probabilities with which we search for an element
        self.q = [0] * self.SIZE  # Probabilities that an element is not found
        self.a = [0] * self.SIZE  # Elements from which OBST is to be built
        self.w = [[0] * self.SIZE for _ in range(self.SIZE)]  # Weight 'w[i][j]' of a tree having root
        self.c = [[0] * self.SIZE for _ in range(self.SIZE)]  # Cost 'c[i][j]' of a tree having root 'r[i][j]'
        self.r = [[0] * self.SIZE for _ in range(self.SIZE)]  # Represents root
        self.n = 0  # Number of nodes

    # This function accepts the input data
    def get_data(self):
        print("\nOptimal Binary Search Tree")
        self.n = int(input("Enter the number of nodes: "))

        print("\nEnter the data:")
        for i in range(1, self.n + 1):
            self.a[i] = int(input(f"a[{i}] "))

        # Input for search probabilities
        for i in range(1, self.n + 1):
            self.p[i] = float(input(f"p[{i}] "))

        # Input for probabilities of not finding the elements
        for i in range(self.n + 1):
            self.q[i] = float(input(f"q[{i}]: "))

    # This function returns a value in the range 'r[i][j-1]' to r[i+1][j]'
    def Min_Value(self, i, j):
        minimum = sys.maxsize
        k = -1
        for m in range(self.r[i][j - 1], self.r[i + 1][j] + 1):
            if self.c[i][m - 1] + self.c[m][j] < minimum:
                minimum = self.c[i][m - 1] + self.c[m][j]
                k = m
        return k

    # This function builds the table from all the given probabilities
    # It basically computes c, r, and w values
    def build_OBST(self):
        # Initialize w, c, and r tables
        for i in range(self.n + 1):
            self.w[i][i] = self.q[i]
            self.r[i][i] = 0
            self.c[i][i] = 0

        # Optimal trees with one node
        for i in range(self.n):
            self.w[i][i + 1] = self.q[i] + self.q[i + 1] + self.p[i + 1]
            self.r[i][i + 1] = i + 1
```

```python
            self.c[i][i + 1] = self.q[i] + self.q[i + 1] + self.p[i + 1]

        self.w[self.n][self.n] = self.q[self.n]
        self.r[self.n][self.n] = 0
        self.c[self.n][self.n] = 0

        # Find optimal trees with 'm' nodes
        for m in range(2, self.n + 1):
            for i in range(self.n - m + 1):
                j = i + m
                self.w[i][j] = self.w[i][j - 1] + self.p[j] + self.q[j]
                k = self.Min_Value(i, j)
                self.c[i][j] = self.w[i][j] + self.c[i][k - 1] + self.c[k][j]
                self.r[i][j] = k

    # This function builds the tree from the tables made by the OBST function
    def build_tree(self):
        queue = []
        front = -1
        rear = -1

        print(f"\nThe Root of this OBST is: {self.r[0][self.n]}")
        print(f"The Cost of this OBST is: {self.c[0][self.n]}")
        print("\n\t NODE \t LEFT CHILD \t RIGHT CHILD")

        # Initializing queue
        queue.append((0, self.n))
        rear += 1

        while front != rear:
            i, j = queue[front + 1]
            front += 1
            k = self.r[i][j]

            print(f"\n\t {k}", end="")

            # Checking for left child
            if self.r[i][k - 1] != 0:
                print(f"\t\t {self.r[i][k - 1]}", end="")
                queue.append((i, k - 1))
                rear += 1
            else:
                print("\t\t", end="")

            # Checking for right child
            if self.r[k][j] != 0:
                print(f"\t {self.r[k][j]}", end="")
                queue.append((k, j))
                rear += 1
            else:
                print("\t", end="")

        print("\n")
```

```
# This is the main function
def main():
    obj = OBST()
    obj.get_data()
    obj.build_OBST()
    obj.build_tree()


if __name__ == "__main__":
    main()
```

OUTPUT

```
Optimal Binary Search Tree
Enter the number of nodes: 3

Enter the data:
a[1] 5
a[2] 2
a[3] 8
p[1] 1
p[2] 1
p[3] 1
q[0]: 0
q[1]: 0
q[2]: 0
q[3]: 0

The Root of this OBST is: 2
The Cost of this OBST is: 5.0

         NODE     LEFT CHILD       RIGHT CHILD

         2                1        3
         1
         3


...Program finished with exit code 0
Press ENTER to exit console.
```

PRACTICAL NO: 7

```
import sys

class Graph:
    def __init__(self, n):
        self.num = n
        self.data = []  # List to store city names
        self.AM = [[0] * n for _ in range(n)]  # Adjacency matrix for the graph

        # Input for city names
        print("Enter names of all cities:")
        for i in range(n):
            self.data.append(input())

        # Input for costs to connect cities
        print("Enter cost to connect cities (enter 0 if no connection):")
        for i in range(n):
            for j in range(n):
                if i == j:
                    self.AM[i][j] = 0  # No cost to connect a city to itself
                else:
                    cost = int(input(f"Cost to connect {self.data[i]} and {self.data[j]}: "))
                    if cost == 0:
                        self.AM[i][j] = sys.maxsize  # No connection if the cost is 0
                    else:
                        self.AM[i][j] = cost

    def prims(self):
        visited = [False] * self.num
        distance = [sys.maxsize] * self.num  # Set all distances to infinity
        from_city = [-1] * self.num  # Array to store the parent city
        total_cost = 0

        distance[0] = 0  # Start from the first city (index 0)

        for _ in range(self.num - 1):
            # Find the unvisited city with the smallest distance
            min_distance = sys.maxsize
            u = -1
            for i in range(self.num):
                if not visited[i] and distance[i] < min_distance:
                    min_distance = distance[i]
                    u = i

            # Mark the selected city as visited
            visited[u] = True
            total_cost += min_distance

            # Update the distances of the neighboring cities
            for v in range(self.num):
                if not visited[v] and self.AM[u][v] != sys.maxsize and self.AM[u][v] < distance[v]:
                    distance[v] = self.AM[u][v]
                    from_city[v] = u
```

```
        # Display the MST
        print("\nCities that need to be connected:")
        for i in range(1, self.num):
            print(f"{self.data[from_city[i]]} -> {self.data[i]} with cost {self.AM[from_city[i]][i]}")

        print(f"Total cost of connecting all cities: {total_cost}")


def main():
    n = int(input("Enter number of cities: "))
    gr = Graph(n)  # Create the graph object for cities
    gr.prims()  # Run Prim's algorithm to find MST


if __name__ == "__main__":
    main()
```

OUTPUT

```
Enter number of cities: 3
Enter names of all cities:
A
B
C
Enter cost to connect cities (enter 0 if no connection):
Cost to connect A and B: 100
Cost to connect A and C: 0
Cost to connect B and A: 200
Cost to connect B and C: 400
Cost to connect C and A: 500
Cost to connect C and B: 0

Cities that need to be connected:
A -> B with cost 100
B -> C with cost 400
Total cost of connecting all cities: 100


...Program finished with exit code 0
Press ENTER to exit console.
```

PRACTICAL NO: 6

```
from collections import deque

# Function to display the adjacency matrix of the graph
def display_graph(cost, n):
    print("The adjacency matrix of the graph is:")
    for i in range(n):
        for j in range(n):
            print(cost[i][j], end=" ")
        print()

# BFS function
def bfs(cost, n, v):
    visited = [False] * n
    queue = deque([v])
    visited[v] = True
    print(f"The BFS of the Graph is:")
    print(v, end=" ")

    while queue:
        v = queue.popleft()
        for j in range(n):
            if cost[v][j] != 0 and not visited[j]:
                visited[j] = True
                queue.append(j)
                print(j, end=" ")

    print()  # for newline

# DFS function
def dfs(cost, n, v):
    visited = [False] * n
    stack = [v]
    print(f"The DFS of the Graph is:")
    print(v, end=" ")

    while stack:
        v = stack.pop()
        if not visited[v]:
            visited[v] = True
            print(v, end=" ")
            for j in range(n-1, -1, -1):  # reverse order to visit neighbors
                if cost[v][j] != 0 and not visited[j]:
                    stack.append(j)
    print()  # for newline

# Main function
def main():
    n = int(input("Enter number of vertices: "))
    m = int(input("Enter number of edges: "))
```

```
    # Initialize the cost (adjacency matrix)
    cost = [[0] * n for _ in range(n)]

    print("\nEDGES")
    for _ in range(m):
        i, j = map(int, input().split())
        cost[i][j] = 1
        cost[j][i] = 1  # Since the graph is undirected

    display_graph(cost, n)

    # BFS
    v = int(input("Enter initial vertex for BFS: "))
    bfs(cost, n, v)

    # DFS
    v = int(input("Enter initial vertex for DFS: "))
    dfs(cost, n, v)

if __name__ == "__main__":
    main()
```

OUTPUT

```
Enter number of vertices: 5
Enter number of edges: 4

EDGES
0 1
1 2
2 3
3 4
The adjacency matrix of the graph is:
0 1 0 0 0
1 0 1 0 0
0 1 0 1 0
0 0 1 0 1
0 0 0 1 0
Enter initial vertex for BFS: 1
The BFS of the Graph is:
1 0 2 3 4
Enter initial vertex for DFS: 2
The DFS of the Graph is:
2 2 1 0 3 4


...Program finished with exit code 0
Press ENTER to exit console.
```

PRACTICAL NO: 5

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

class ExpressionTree:
    def __init__(self):
        self.root = None

    # Function to construct expression tree from prefix expression
    def construct_tree(self, prefix):
        stack = []
        for char in reversed(prefix):  # Traverse from right to left
            if char.isalpha():  # Operand (e.g., 'a', 'b', 'c')
                stack.append(Node(char))
            else:  # Operator (e.g., '+', '-', '*', '/')
                node = Node(char)
                node.left = stack.pop()  # Pop two operands from stack
                node.right = stack.pop()
                stack.append(node)  # Push the subtree back to stack

        # The root of the tree will be the last remaining node
        self.root = stack[-1]

    # Non-recursive post-order traversal
    def post_order_traversal(self):
        if not self.root:
            return

        stack1 = [self.root]
        stack2 = []

        while stack1:
            node = stack1.pop()
            stack2.append(node)

            # Push left and right children to stack1
            if node.left:
                stack1.append(node.left)
            if node.right:
                stack1.append(node.right)

        # Print nodes in post-order
        while stack2:
            print(stack2.pop().data, end=" ")

    # Delete the entire tree (simulated by clearing the root reference)
    def delete_tree(self):
        self.root = None
        print("\nTree deleted.")

# Main program
```

```
if __name__ == "__main__":
    # Take input for prefix expression
    prefix_expression = input("Enter a prefix expression: ")

    # Create expression tree and construct it
    tree = ExpressionTree()
    tree.construct_tree(prefix_expression)

    # Perform non-recursive post-order traversal
    print("Post-order Traversal (Non-Recursive):")
    tree.post_order_traversal()  # Should print post-order of the expression tree

    # Delete the entire tree
    tree.delete_tree()
```

OUTPUT

```
Enter a prefix expression: --a*bc/def
Post-order Traversal (Non-Recursive):
a b c * - d e / -
Tree deleted.


...Program finished with exit code 0
Press ENTER to exit console.
```

PRACTICAL NO: 4

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

class BinarySearchTree:
    def __init__(self):
        self.root = None

    # Function to insert a node into the BST
    def insert(self, data):
        if not self.root:
            self.root = Node(data)
        else:
            self._insert_recursive(self.root, data)

    # Helper function for recursive insertion
    def _insert_recursive(self, node, data):
        if data < node.data:
            if node.left is None:
                node.left = Node(data)
            else:
                self._insert_recursive(node.left, data)
        else:
            if node.right is None:
                node.right = Node(data)
            else:
                self._insert_recursive(node.right, data)

    # Function to insert a new node
    def insert_new_node(self, data):
        self.insert(data)

    # Function to find the height of the tree (Longest path from root)
    def find_longest_path(self):
        return self._height(self.root)

    # Helper function for calculating height
    def _height(self, node):
        if node is None:
            return 0
        left_height = self._height(node.left)
        right_height = self._height(node.right)
        return max(left_height, right_height) + 1

    # Function to find the minimum data value in the tree
    def find_min_value(self):
        return self._find_min(self.root)

    # Helper function to find the minimum value
    def _find_min(self, node):
        current = node
```

```
        while current.left is not None:
            current = current.left
        return current.data

    # Function to swap left and right at every node
    def swap_left_right(self):
        self._swap_recursive(self.root)

    # Helper function to recursively swap left and right pointers
    def _swap_recursive(self, node):
        if node is not None:
            node.left, node.right = node.right, node.left
            self._swap_recursive(node.left)
            self._swap_recursive(node.right)

    # Function to search for a value in the tree
    def search(self, value):
        return self._search_recursive(self.root, value)

    # Helper function for recursive search
    def _search_recursive(self, node, value):
        if node is None:
            return False
        if node.data == value:
            return True
        elif value < node.data:
            return self._search_recursive(node.left, value)
        else:
            return self._search_recursive(node.right, value)

    # Function to print the tree for testing purposes
    def print_tree(self):
        self._print_recursive(self.root)

    # Helper function for in-order traversal (for printing the tree)
    def _print_recursive(self, node):
        if node is not None:
            self._print_recursive(node.left)
            print(node.data, end=" ")
            self._print_recursive(node.right)

# Main program
if __name__ == "__main__":
    bst = BinarySearchTree()

    # Take input for tree construction
    n = int(input("Enter number of values to insert into the tree: "))
    values = [int(input(f"Enter value {i + 1}: ")) for i in range(n)]

    for value in values:
        bst.insert(value)

    while True:
        # Menu for operation selection
        print("\n--- Menu ---")
```

```
    print("Press 1 for Insert new node")
    print("Press 2 for Find number of nodes in longest path from root")
    print("Press 3 for Minimum data value found in the tree")
    print("Press 4 for Change a tree so that the roles of the left and right pointers are swapped at every 
node")
    print("Press 5 for Search a value")
    print("Press 6 to Exit")

    choice = input("Enter your choice: ")

    if choice == '1':
        # Insert new node
        new_node = int(input("\nEnter a new value to insert into the tree: "))
        bst.insert_new_node(new_node)
        print("\nTree after inserting the new node:")
        bst.print_tree()

    elif choice == '2':
        # Find number of nodes in the longest path from root
        print(f"\nLongest path from root (Height of tree): {bst.find_longest_path()}")

    elif choice == '3':
        # Find the minimum data value in the tree
        print(f"Minimum value in the tree: {bst.find_min_value()}")

    elif choice == '4':
        # Swap left and right at every node
        bst.swap_left_right()
        print("\nTree after swapping left and right at every node:")
        bst.print_tree()

    elif choice == '5':
        # Search for a value
        value_to_search = int(input("\nEnter a value to search for in the tree: "))
        found = bst.search(value_to_search)
        print(f"Is {value_to_search} found in the tree? {found}")

    elif choice == '6':
        # Exit the program
        print("Exiting the program...")
        break

    else:
        print("Invalid choice, please try again.")
```

OUTPUT

```
Enter number of values to insert into the tree: 4
Enter value 1: 12
Enter value 2: 13
Enter value 3: 455
Enter value 4: 67

--- Menu ---
Press 1 for Insert new node
Press 2 for Find number of nodes in longest path from root
Press 3 for Minimum data value found in the tree
Press 4 for Change a tree so that the roles of the left and right pointers are swapped at every node
Press 5 for Search a value
Press 6 to Exit
Enter your choice: 1

Enter a new value to insert into the tree: 900

Tree after inserting the new node:
12 13 67 455 900
--- Menu ---
Press 1 for Insert new node
Press 2 for Find number of nodes in longest path from root
Press 3 for Minimum data value found in the tree
Press 4 for Change a tree so that the roles of the left and right pointers are swapped at every node
Press 5 for Search a value
Press 6 to Exit
Enter your choice: 5

Enter a value to search for in the tree: 13
Is 13 found in the tree? True

--- Menu ---
Press 1 for Insert new node
Press 2 for Find number of nodes in longest path from root
Press 3 for Minimum data value found in the tree
Press 4 for Change a tree so that the roles of the left and right pointers are swapped at every node
Press 5 for Search a value
Press 6 to Exit
Enter your choice: 2

Longest path from root (Height of tree): 4

--- Menu ---
Press 1 for Insert new node
Press 2 for Find number of nodes in longest path from root
Press 3 for Minimum data value found in the tree
Press 4 for Change a tree so that the roles of the left and right pointers are swapped at every node
Press 5 for Search a value
Press 6 to Exit
Enter your choice: 3
Minimum value in the tree: 12

--- Menu ---
Press 1 for Insert new node
Press 2 for Find number of nodes in longest path from root
Press 3 for Minimum data value found in the tree
Press 4 for Change a tree so that the roles of the left and right pointers are swapped at every node
Press 5 for Search a value
Press 6 to Exit
Enter your choice: 4

Tree after swapping left and right at every node:
900 455 67 13 12
--- Menu ---
Press 1 for Insert new node
Press 2 for Find number of nodes in longest path from root
Press 3 for Minimum data value found in the tree
Press 4 for Change a tree so that the roles of the left and right pointers are swapped at every node
Press 5 for Search a value
Press 6 to Exit
Enter your choice: 6
Exiting the program...


...Program finished with exit code 0
Press ENTER to exit console.
```

PRACTICAL NO: 3

```
class TreeNode:
    def _init_(self, name):
        self.name = name
        self.children = []

    def add_child(self, child):
        self.children.append(child)

    def print_tree(self, level=0):
        # Print the current node with indentation based on its level
        print("  " * level + self.name)
        # Recursively print each child
        for child in self.children:
            child.print_tree(level + 1)


# Function to build the tree from user input
def build_book_tree():
    # Create the root node (Book)
    book = TreeNode("Book")

    # Input chapters
    num_chapters = int(input("Enter the number of chapters: "))
    for i in range(1, num_chapters + 1):
        chapter_name = input(f"Enter the name of Chapter {i}: ")
        chapter = TreeNode(chapter_name)
        book.add_child(chapter)

        # Input sections for this chapter
        num_sections = int(input(f"Enter the number of sections in {chapter_name}: "))
        for j in range(1, num_sections + 1):
            section_name = input(f"Enter the name of Section {j} in {chapter_name}: ")
            section = TreeNode(section_name)
            chapter.add_child(section)

            # Input subsections for this section
            num_subsections = int(input(f"Enter the number of subsections in {section_name}: "))
            for k in range(1, num_subsections + 1):
                subsection_name = input(f"Enter the name of Subsection {k} in {section_name}: ")
                subsection = TreeNode(subsection_name)
                section.add_child(subsection)

    return book


# Main program
if _name_ == "_main_":

    book_tree = build_book_tree()
    print("\nBook Structure:")
    book_tree.print_tree()
```

OUTPUT

```
Enter the number of chapters: 1
Enter the name of Chapter 1: Wings of Fire
Enter the number of sections in Wings of Fire: 2
Enter the name of Section 1 in Wings of Fire: INTRODUCTION
Enter the number of subsections in INTRODUCTION: 1
Enter the name of Subsection 1 in INTRODUCTION: Things About Kalam
Enter the name of Section 2 in Wings of Fire: Conclusion of kalam
Enter the number of subsections in Conclusion of kalam: 1
Enter the name of Subsection 1 in Conclusion of kalam: Summary

Book Structure:
Book
  Wings of Fire
    INTRODUCTION
      Things About Kalam
    Conclusion of kalam
      Summary


...Program finished with exit code 0
Press ENTER to exit console.
```

PRACTICAL NO: 2

```
class HashFunction:
    class Hash:
        def __init__(self):
            self.key = -1
            self.name = "NULL"

    def __init__(self):
        self.h = [self.Hash() for _ in range(10)]

    def insert(self):
        cnt = 0
        while cnt < 10:
            if cnt >= 10:
                print("\n\tHash Table is FULL")
                break

            k = int(input("\n\tEnter a Telephone No: "))
            n = input("\n\tEnter a Client Name: ")

            hi = k % 10  # hash function
            if self.h[hi].key == -1:
                self.h[hi].key = k
                self.h[hi].name = n
            else:
                # Collision resolution using linear probing
                flag = False
                if self.h[hi].key % 10 != hi:
                    temp = self.h[hi].key
                    ntemp = self.h[hi].name
                    self.h[hi].key = k
                    self.h[hi].name = n
                    for i in range(hi + 1, 10):
                        if self.h[i].key == -1:
                            self.h[i].key = temp
                            self.h[i].name = ntemp
                            flag = True
                            break
                    for i in range(0, hi):
                        if self.h[i].key == -1:
                            self.h[i].key = temp
                            self.h[i].name = ntemp
                            break
                else:
                    for i in range(hi + 1, 10):
                        if self.h[i].key == -1:
                            self.h[i].key = k
                            self.h[i].name = n
                            flag = True
                            break
                    for i in range(0, hi):
                        if self.h[i].key == -1:
                            self.h[i].key = k
                            self.h[i].name = n
```

```
                    break
            cnt += 1
            ans = input("\n\t..... Do You Want to Insert More Key: y/n")
            if ans.lower() != 'y':
                break

    def display(self):
        print("\n\t\tKey\t\tName")
        for i in range(10):
            print(f"\n\th[{i}]\t{self.h[i].key}\t\t{self.h[i].name}")

    def find(self, k):
        for i in range(10):
            if self.h[i].key == k:
                print(f"\n\t{k} is Found at {i} Location With Name {self.h[i].name}")
                return i
        return -1

    def delete(self, k):
        index = self.find(k)
        if index == -1:
            print("\n\tKey Not Found")
        else:
            self.h[index].key = -1
            self.h[index].name = "NULL"
            print("\n\tKey is Deleted")


if __name__ == "__main__":
    obj = HashFunction()
    while True:
        print("\n\t*** Telephone (ADT) ***")
        print("\n\t1. Insert\n\t2. Display\n\t3. Find\n\t4. Delete\n\t5. Exit")
        ch = int(input("\n\t..... Enter Your Choice: "))

        if ch == 1:
            obj.insert()
        elif ch == 2:
            obj.display()
        elif ch == 3:
            k = int(input("\n\tEnter a Key Which You Want to Search: "))
            index = obj.find(k)
            if index == -1:
                print("\n\tKey Not Found")
        elif ch == 4:
            k = int(input("\n\tEnter a Key Which You Want to Delete: "))
            obj.delete(k)
        elif ch == 5:
            break

        ans = input("\n\t..... Do You Want to Continue in Main Menu: y/n ")
        if ans.lower() != 'y':
            break
```

OUTPUT

```
*** Telephone (ADT) ***

1. Insert
2. Display
3. Find
4. Delete
5. Exit

..... Enter Your Choice: 1

Enter a Telephone No: 12345

Enter a Client Name: Shreyash

..... Do You Want to Insert More Key: y/ny

Enter a Telephone No: 112233

Enter a Client Name: Tanishkk

..... Do You Want to Insert More Key: y/ny

Enter a Telephone No: 667788

Enter a Client Name: Ishan

..... Do You Want to Insert More Key: y/ny

Enter a Telephone No: 0000

Enter a Client Name: Ram

..... Do You Want to Insert More Key: y/nn

..... Do You Want to Continue in Main Menu: y/n y

*** Telephone (ADT) ***

1. Insert
2. Display
3. Find
4. Delete
5. Exit
```

```
..... Enter Your Choice: 2

        Key             Name

h[0]    0               Ram

h[1]    -1              NULL

h[2]    -1              NULL

h[3]    112233          Tanishkk

h[4]    -1              NULL

h[5]    12345           Shreyash

h[6]    -1              NULL

h[7]    -1              NULL

h[8]    667788          Ishan

h[9]    -1              NULL

..... Do You Want to Continue in Main Menu: y/n y

*** Telephone (ADT) ***

1. Insert
2. Display
3. Find
4. Delete
5. Exit

..... Enter Your Choice: 3

Enter a Key Which You Want to Search: 12345

12345 is Found at 5 Location With Name Shreyash

..... Do You Want to Continue in Main Menu: y/n y

*** Telephone (ADT) ***
```

```
1. Insert
2. Display
3. Find
4. Delete
5. Exit

..... Enter Your Choice: 3

Enter a Key Which You Want to Search: 12345

12345 is Found at 5 Location With Name Shreyash

..... Do You Want to Continue in Main Menu: y/n y

*** Telephone (ADT) ***

1. Insert
2. Display
3. Find
4. Delete
5. Exit

..... Enter Your Choice: 2

        Key             Name

h[0]    0               Ram

h[1]    -1              NULL

h[2]    -1              NULL

h[3]    112233          Tanishkk

h[4]    -1              NULL

h[5]    12345           Shreyash

h[6]    -1              NULL

h[7]    -1              NULL

h[8]    667788          Ishan

h[9]    -1              NULL
```

```
..... Enter Your Choice: 4

Enter a Key Which You Want to Delete: 0000

0 is Found at 0 Location With Name Ram

Key is Deleted

..... Do You Want to Continue in Main Menu: y/n y

*** Telephone (ADT) ***

1. Insert
2. Display
3. Find
4. Delete
5. Exit

..... Enter Your Choice: 2

        Key             Name

h[0]    -1              NULL

h[1]    -1              NULL

h[2]    -1              NULL

h[3]    112233          Tanishkk

h[4]    -1              NULL

h[5]    12345           Shreyash

h[6]    -1              NULL

h[7]    -1              NULL

h[8]    667788          Ishan

h[9]    -1              NULL

..... Do You Want to Continue in Main Menu: y/n y

*** Telephone (ADT) ***

1. Insert
2. Display
3. Find
4. Delete
5. Exit

..... Enter Your Choice: 5

...Program finished with exit code 0
Press ENTER to exit console.
```

PRACTICAL NO: 1

```
class hashTable:
   # initialize hash Table
   def __init__(self):
      self.size = int(input("Enter the Size of the hash table: "))
      # initialize table with all elements None
      self.table = list(None for i in range(self.size))
      self.elementCount = 0
      self.comparisons = 0

   # method that checks if the hash table is full or not
   def isFull(self):
      if self.elementCount == self.size:
         return True
      else:
         return False

   # method that returns position for a given element
   def hashFunction(self, element):
      return element % self.size

   # method that inserts element into the hash table
   def insert(self, record):
      # checking if the table is full
      if self.isFull():
         print("Hash Table Full")
         return False

      isStored = False
      position = self.hashFunction(record.get_number())

      # checking if the position is empty
      if self.table[position] == None:
         self.table[position] = record
         print("Phone number of " + record.get_name() + " is at position " + str(position))
         isStored = True
         self.elementCount += 1
      # collision occurred hence we do linear probing
      else:
         print("Collision has occurred for " + record.get_name() + "'s phone number at position " +
str(position) + " finding new Position.")
         while self.table[position] != None:
            position += 1
            if position >= self.size:
               position = 0
         self.table[position] = record
         print("Phone number of " + record.get_name() + " is at position " + str(position))
         isStored = True
         self.elementCount += 1

      return isStored

   # method that searches for an element in the table
   # returns position of element if found, else returns False
```

```
    def search(self, record):
        found = False
        position = self.hashFunction(record.get_number())
        self.comparisons += 1
        if(self.table[position] != None):
            if(self.table[position].get_name() == record.get_name() and self.table[position].get_number() == record.get_number()):
                isFound = True
                print("Phone number found at position {0} ".format(position) + " and total comparisons are " + str(1))
                return position
        # if element is not found at position returned by hash function
        else:
            position += 1
            if position >= self.size - 1:
                position = 0
            while self.table[position] != None or self.comparisons <= self.size:
                if(self.table[position].get_name() == record.get_name() and self.table[position].get_number() == record.get_number()):
                    isFound = True
                    i = self.comparisons + 1
                    print("Phone number found at position {0} ".format(position) + " and total comparisons are " + str(i))
                    return position
                position += 1
                if position >= self.size - 1:
                    position = 0
                self.comparisons += 1
            if isFound == False:
                print("Record not found")
                return False

    # method to display the hash table
    def display(self):
        print("\n")
        for i in range(self.size):
            print("Hash Value: " + str(i) + "\t\t" + str(self.table[i]))
        print("The number of phonebook records in the Table are: " + str(self.elementCount))


class Record:
    def __init__(self):
        self._name = None
        self._number = None

    def get_name(self):
        return self._name

    def get_number(self):
        return self._number

    def set_name(self, name):
        self._name = name

    def set_number(self, number):
```

```
        self._number = number

    def __str__(self):
        record = "Name: " + str(self.get_name()) + "\t" + "\tNumber: " + str(self.get_number())
        return record


class doubleHashTable:
    # initialize hash Table
    def __init__(self):
        self.size = int(input("Enter the Size of the hash table: "))
        # initialize table with all elements None
        self.table = list(None for i in range(self.size))
        self.elementCount = 0
        self.comparisons = 0

    # method that checks if the hash table is full or not
    def isFull(self):
        if self.elementCount == self.size:
            return True
        else:
            return False

    # First hash function
    def h1(self, element):
        return element % self.size

    # Second hash function
    def h2(self, element):
        return 5 - (element % 5)

    # method to resolve collision by double hashing method
    def doubleHashing(self, record):
        posFound = False
        limit = self.size
        i = 1
        # start a loop to find the position
        while i <= limit:
            # calculate new position by double hashing
            newPosition = (self.h1(record.get_number()) + i * self.h2(record.get_number())) % self.size
            # if newPosition is empty then break out of loop and return new Position
            if self.table[newPosition] == None:
                posFound = True
                break
            else:
                # as the position is not empty increase i
                i += 1
        return posFound, newPosition

    # method that inserts element inside the hash table
    def insert(self, record):
        # checking if the table is full
        if self.isFull():
            print("Hash Table Full")
            return False
```

```
        posFound = False
        position = self.h1(record.get_number())

        # checking if the position is empty
        if self.table[position] == None:
            self.table[position] = record
            print("Phone number of " + record.get_name() + " is at position " + str(position))
            posFound = True
            self.elementCount += 1
        # If collision occurred
        else:
            print("Collision has occurred for " + record.get_name() + "'s phone number at position " +
str(position) + " finding new Position.")
            while not posFound:
                posFound, position = self.doubleHashing(record)
            if posFound:
                self.table[position] = record
                self.elementCount += 1
                print("Phone number of " + record.get_name() + " is at position " + str(position))

        return posFound

    # searches for an element in the table and returns position of element if found else returns False
    def search(self, record):
        found = False
        position = self.h1(record.get_number())
        self.comparisons += 1

        if(self.table[position] != None):
            if(self.table[position].get_name() == record.get_name() and self.table[position].get_number() ==
record.get_number()):
                found = True
                print("Phone number found at position {0} ".format(position) + " and total comparisons are " +
str(self.comparisons))
                return position

        limit = self.size
        i = 1
        # start a loop to find the position
        while i <= limit:
            position = (self.h1(record.get_number()) + i * self.h2(record.get_number())) % self.size
            self.comparisons += 1
            # if element at newPosition is equal to the required element
            if(self.table[position] != None):
                if self.table[position].get_name() == record.get_name():
                    found = True
                    break
            elif self.table[position].get_name() == None:
                found = False
                break
            else:
                # as the position is not empty increase i
                i += 1
```

```
        if found:
            print("Phone number found at position {}".format(position) + " and total comparisons are " + str(i +
1))
        else:
            print("Record not Found")
        return found

    # method to display the hash table
    def display(self):
        print("\n")
        for i in range(self.size):
            print("Hash Value: " + str(i) + "\t\t" + str(self.table[i]))
        print("The number of phonebook records in the Table are: " + str(self.elementCount))


def input_record():
    record = Record()
    name = input("Enter Name:")
    number = int(input("Enter Number:"))
    record.set_name(name)
    record.set_number(number)
    return record


choice1 = 0
while choice1 != 3:
    print("*")
    print("1. Linear Probing *")
    print("2. Double Hashing *")
    print("3. Exit *")
    print("*")
    choice1 = int(input("Enter Choice: "))
    if choice1 > 3:
        print("Please Enter Valid Choice")

    if choice1 == 1:
        h1 = hashTable()
        choice2 = 0
        while choice2 != 4:
            print("*")
            print("1. Insert *")
            print("2. Search *")
            print("3. Display *")
            print("4. Back *")
            print("*")
            choice2 = int(input("Enter Choice: "))
            if choice2 > 4:
                print("Please Enter Valid Choice")

            if choice2 == 1:
                record = input_record()
                h1.insert(record)
            elif choice2 == 2:
                record = input_record()
                position = h1.search(record)
```

```
        elif choice2 == 3:
            h1.display()

    elif choice1 == 2:
        h2 = doubleHashTable()
        choice2 = 0
        while choice2 != 4:
            print("*")
            print("1. Insert *")
            print("2. Search *")
            print("3. Display *")
            print("4. Back *")
            print("*")
            choice2 = int(input("Enter Choice: "))
            if choice2 > 4:
                print("Please Enter Valid Choice")

            if choice2 == 1:
                record = input_record()
                h2.insert(record)
            elif choice2 == 2:
                record = input_record()
                position = h2.search(record)
            elif choice2 == 3:
                h2.display()
```

OUTPUT

```
*
1. Linear Probing *
2. Double Hashing *
3. Exit *
*
Enter Choice: 1
Enter the Size of the hash table: 6
*
1. Insert *
2. Search *
3. Display *
4. Back *
*
Enter Choice: 1
Enter Name:Shreyash
Enter Number:123
Phone number of Shreyash is at position 3
*
1. Insert *
2. Search *
3. Display *
4. Back *
*
Enter Choice: 1
Enter Name:Varad
Enter Number:456
Phone number of Varad is at position 0
*
1. Insert *
2. Search *
3. Display *
4. Back *
*
Enter Choice: 1
Enter Name:Ishan
Enter Number:789
Collision has occurred for Ishan's phone number at position 3 finding new Position.
Phone number of Ishan is at position 4
*
1. Insert *
2. Search *
3. Display *
4. Back *
*
Enter Choice: 3


Hash Value: 0           Name: Varad             Number: 456
Hash Value: 1           None
Hash Value: 2           None
Hash Value: 3           Name: Shreyash          Number: 123
Hash Value: 4           Name: Ishan             Number: 789
Hash Value: 5           None
The number of phonebook records in the Table are: 3
```

```
*
1. Insert *
2. Search *
3. Display *
4. Back *
*
Enter Choice: 3


Hash Value: 0           Name: Shreyash          Number: 123
Hash Value: 1           Name: Varad             Number: 456
Hash Value: 2           Name: Ishan             Number: 788
The number of phonebook records in the Table are: 3
*
1. Insert *
2. Search *
3. Display *
4. Back *
*
Enter Choice: 2
Enter Name:Varad
Enter Number:456
Phone number found at position 1 and total comparisons are 2
*
1. Insert *
2. Search *
3. Display *
4. Back *
*
Enter Choice: 4
*
1. Linear Probing *
2. Double Hashing *
3. Exit *
*
Enter Choice:
```

```
*
1. Insert *
2. Search *
3. Display *
4. Back *
*
Enter Choice: 2
Enter Name:Shreyash
Enter Number:123
Phone number found at position 3  and total comparisons are 1
*
1. Insert *
2. Search *
3. Display *
4. Back *
*
Enter Choice: 4
*
1. Linear Probing *
2. Double Hashing *
3. Exit *
*
Enter Choice: 2
Enter the Size of the hash table: 3
*
1. Insert *
2. Search *
3. Display *
4. Back *
*
Enter Choice: 1
Enter Name:Shreyash
Enter Number:123
Phone number of Shreyash is at position 0
*
1. Insert *
2. Search *
3. Display *
4. Back *
*
Enter Choice: 1
Enter Name:Varad
Enter Number:456
Collision has occurred for Varad's phone number at position 0 finding new Position.
Phone number of Varad is at position 1
*
1. Insert *
2. Search *
3. Display *
4. Back *
*
Enter Choice: 1
Enter Name:Ishan
Enter Number:788
Phone number of Ishan is at position 2
```