AIM OF THE EXPERIMENT:-

Study different types of data structures using python programing language.

EXPERIMENT: 1 [List & Tuples]

Create a python list containing the names of five countries.

```
Country = ["India", "Australia", "Africa", "America", "England"]
print("Before append: ", Country)
```

Perform the following operation.

a) Add two more countries to the list.

```
Country.append("Pakistan")
Country.append("Russia")
print("After append: ", Country)
```

b) Print the third country in the list.

```
print("Third country: ", Country[2])
```

c) Sort the list in alphabetical order.

```
Sorted_List = sorted(Country)
print("Sorted List: ", Sorted_List)
```

d) Check if the country 'India' is present in the list.

```
if "India" in Country:
    print("Country Exists in the list.")
```

OUTPUT:-

```
Before append:  ['India', 'Australia', 'Africa', 'America', 'England']
After append:  ['India', 'Australia', 'Africa', 'America', 'England', 'Pakistan', 'Russia']
Third country:  Africa
Sorted List:  ['Africa', 'America', 'Australia', 'England', 'India', 'Pakistan', 'Russia']
Country Exists in the list.
```

Create a tuple with five elements, each representing the price of a product.

```
Price = (1050, 2500, 758, 15000, 8000)
```

Perform the following operation.

a) Find the maximum and minimum price from the tuple.

```
print("Maximum price: ", max(Price))
print("Minimum price: ", min(Price))
```

b) Calculate the total cost of all the products.

```
print("Total cost: ", sum(Price))
```

c) Convert the tuple to a list and add a new product with its price.

```
Price_List = list(Price)
print("List= ", Price_List)
Price_List.append(23000)
print("After append: ", Price_List)
```

d) Calculate the average price of the products.

```
print("Average: ", sum(Price_List)/len(Price_List))
```

OUTPUT:-

```
Maximum price:  15000
Minimum price:  758
Total cost:  27308
List=  [1050, 2500, 758, 15000, 8000]
After append:  [1050, 2500, 758, 15000, 8000, 23000]
Average:  8384.666666666666
```

EXPERIMENT: 2 [Dictionaries]

Create a dictionary that represents the population of five cities.

```
Dict = {"City1": 65000, "City2": 70000, "London": 55000, "City4":
100000, "City5": 150000}
print("Dictionary: ", Dict)
```

Perform the following operation.

a) Add two more cities and their population to the dictionary.

```
Dict.update({"City6": 23000, "City7": 49000})
print("Dict After update: ", Dict)
```

b) Find the city with highest population.

```
print("Maximum Population: ", max(Dict.items(), key=lambda x:
x[1]))
```

c) Check if a city 'London' is present in the dictionary.

```
if "London" in Dict:
    print("City Exist")
```

d) Remove a city and its population from dictionary.

```
Dict.pop("City6")
print("New Dict: ", Dict)
```

OUTPUT:-

```
Dictionary:  {'City1': 65000, 'City2': 70000, 'London': 55000, 'City4': 100000, 'City5': 150000}
Dict After update:  {'City1': 65000, 'City2': 70000, 'London': 55000, 'City4': 100000, 'City5': 150000, 'City6': 23000, 'City7': 49000}
Maximum Population:  ('City5', 150000)
City Exist
New Dict:  {'City1': 65000, 'City2': 70000, 'London': 55000, 'City4': 100000, 'City5': 150000, 'City7': 49000}
```

Create a dictionary that maps names of students to their respective marks in an exam.

```
Student = {"Shankar": 98, "Biswajit": 87, "Prince": 63}
print("Student List: ", Student)
```

Perform the following operation.

a) Add a new student and their marks to the dictionary.

```
Student.update({"Aman": 75})
print("Updated Student List: ", Student)
```

b) Calculate the average marks of all the students.

```
print("Average marks: ", sum(Student.values()) /
len(Student.values()))
```

c) Find the student with the highest marks.

```
print("Highest marks: ", max(Student.items(), key=lambda x:
x[1]))
```

d) Sort the dictionary by student names in alphabetical order.

```
Sorted_Dict = sorted(Student)
print("Sorted Dictionary: ", Sorted_Dict)
```

OUTPUT:-

```
Student List:  {'Shankar': 98, 'Biswajit': 87, 'Prince': 63}
Updated Student List:  {'Shankar': 98, 'Biswajit': 87, 'Prince': 63, 'Aman': 75}
Average marks:  80.75
Highest marks:  ('Shankar', 98)
Sorted Dictionary:  ['Aman', 'Biswajit', 'Prince', 'Shankar']
```

EXPERIMENT: 3 [Stacks & Queues]

Implement a stack using a python list to perform the following operations.

```
stack = []
print("Stack: ", stack)
```

a) Push five elements onto the stack.

```
stack.append(8)
stack.append(2)
stack.append(6)
stack.append(0)
stack.append(5)
print("Stack: ", stack)
```

b) Pop two elements from the stack.

```
print(stack.pop(), " is popped.")
print(stack.pop(), " is popped.")        #LIFO
print("Updated Stack: ", stack)
```

c) Check if the stack is empty.

```
if stack == []:
    print("True: Stack is empty.")
else:
    print("False: Stack is not empty.")
```

OUTPUT:-

```
Stack:  []
Stack:  [8, 2, 6, 0, 5]
5  is popped.
0  is popped.
Updated Stack:  [8, 2, 6]
False: Stack is not empty.
```

Implement a queue using a python list to perform the following operation.

```
queue = []
print("Queue: ", queue)
```

a) Enqueue five elements into the queue.

```
queue.append(2)
queue.append(1)
queue.append(7)
queue.append(8)
queue.append(5)
print("Queue: ", queue)
```

b) Dequeue three elements from the queue.

```
print(queue.pop(0), " is popped")
print(queue.pop(0), " is popped")        # FIFO
print(queue.pop(0), " is popped")
print("Updated queue: ", queue)
```

c) Check if the queue is empty.

```
if queue == []:
    print("True: queue is empty.")
else:
    print("False: queue is not empty.")
```

OUTPUT:-

```
Queue:  []
Queue:  [2, 1, 7, 8, 5]
2  is popped
1  is popped
7  is popped
Updated queue:  [8, 5]
False: queue is not empty.
```

EXPERIMENT: 4 [Linked Lists]

Implement a singly linked list in python to perform the following operations.

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


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

    def print_LL(self):
        if self.head is None:
            print("Linked List is empty!")
        else:
            n = self.head
```

```
            while n is not None:
                print(n.data, "--->", end=" ")
                n = n.ref

    def add_begin(self, data):
        new_node = Node(data)
        new_node.ref = self.head
        self.head = new_node

    def delete_begin(self):
        if self.head is None:
            print("LL is empty. So, can't delete nodes.")
        else:
            self.head = self.head.ref

    def search(self, ll, key):
        if self.head is None:
            return False
        if ll.data == key:
            return True
        return self.search(ll.ref, key)

    @staticmethod
    def reversePrint(head):
        if head is None:
            return
        LinkedList.reversePrint(head.ref)
        print(head.data, "--->", end=" ")


LL1 = LinkedList()
```

a) Insert five elements at the beginning of the list.

```
LL1.add_begin(10)
LL1.add_begin(25)
LL1.add_begin(49)
LL1.add_begin(63)
LL1.add_begin(98)  # 10 will be printed at last due to
insertion at beginning.
print("Created Linked List is:-")
LL1.print_LL()
```

b) Delete an element from the list.

```
LL1.delete_begin()
print("\nUpdated Linked List is:-")
LL1.print_LL()
```

c) Search for a specific element in the list.

```
value = 49
found = LL1.search(LL1.head, value)
if found:
    print(f"\n{value} is present in the linked list.")
else:
    print(f"\n{value} is not present in the linked list.")
```

d) Print the elements of the list in reverse order.

```
print("Linked List in reverse order is:-")
LL1.reversePrint(LL1.head)
```

OUTPUT:-

```
Created Linked List is:-
98 ---> 63 ---> 49 ---> 25 ---> 10 --->
Updated Linked List is:-
63 ---> 49 ---> 25 ---> 10 --->
49 is present in the linked list.
Linked List in reverse order is:-
10 ---> 25 ---> 49 ---> 63 --->
```

EXPERIMENT: 5 [Trees]

Implement a binary search tree in python to perform the following operations.

```
class BST:
    def __init__(self, key):
        self.key = key
        self.lChild = None
        self.rChild = None

    def insert(self, data):
        if self.key is None:
            self.key = data
            return
        if self.key == data:
            return
        if self.key > data:
            if self.lChild:
                self.lChild.insert(data)
            else:
                self.lChild = BST(data)
        else:
            if self.rChild:
                self.rChild.insert(data)
            else:
                self.rChild = BST(data)

    def search(self, data):
        if self.key is None:
            print("Tree is empty!")
            return False
        if self.key == data:
            return True
        if self.key > data:
            if self.lChild:
                return self.lChild.search(data)
            else:
                return False
        else:
            if self.rChild:
                return self.rChild.search(data)
```

```
            else:
                return False

    def __delitem__(self, key):
        if self.key is None:
            print("Tree is empty!")
            return
        if key < self.key:
            if self.lChild:
                self.lChild = self.lChild.__delitem__(key)

            else:
                print("Given node is not present in the tree.")
        elif key > self.key:
            if self.rChild:
                self.rChild = self.rChild.__delitem__(key)
            else:
                print("Given node is not present in the tree.")
        else:
            if self.lChild is None:
                temp = self.rChild
                return temp
            if self.rChild is None:
                temp = self.lChild
                return temp
            node = self.rChild
            while node.lChild:
                node = node.lChild
            # Replace the current node with the successor node
            node = node
            # Delete the successor node from the right subtree
            del node.rChild[node.key]
        return self

    def inorder(self):
        if self.lChild:
            self.lChild.inorder()
        print(self.key, end=" ")
        if self.rChild:
            self.rChild.inorder()

    def height(self):
        # Base case: empty subtree has height 0
        if self is None:
            return 0
        # Recursive case: non-empty subtree has height 1 + max of
left and right subtrees
        else:
            # Check if left child is None
            if self.lChild is None:
                left_height = 0
            else:
                left_height = self.lChild.height()
            # Check if right child is None
            if self.rChild is None:
                right_height = 0
```

```
            else:
                right_height = self.rChild.height()
            return 1 + max(left_height, right_height)


root = BST(25)
```

a) Insert five elements into the tree

```
list1 = [10, 33, 17, 49]
for i in list1:
    root.insert(i)
print("The root node's key is", root.key)
print("After inserting 5 elements into the tree, the inorder
traversal is:")
root.inorder()
print()
```

b) Search for a specific element in the tree.

```
print("Searching for 33 in the tree returns", root.search(33))
```

c) Delete an element from the tree.

```
del root[17]
```

d) Print the tree using in-order traversal.

```
print("After deleting an element from the tree, the inorder
traversal is:")
root.inorder()
print()
```

e) Find the height of the binary search tree.

```
print("The height of the binary search tree is:",
root.height())
```

OUTPUT:-

```
The root node's key is 25
After inserting 5 elements into the tree, the inorder traversal is:
10 17 25 33 49
Searching for 33 in the tree returns True
After deleting an element from the tree, the inorder traversal is:
10 25 33 49
The height of the binary search tree is: 3
```

Submitted By,

Shankar Singh Mahanty
2101020758
CSE21238
Group – 3
Sem – 5th