MALAD KANDIVALI EDUCATION SOCIETY'S

NAGINDAS KHANDWALA COLLEGE OF COMMERCE, ARTS & MANAGEMENT STUDIES & SHANTABEN NAGINDAS KHANDWALA COLLEGE OF SCIENCE

MALAD [W], MUMBAI – 64

AUTONOMOUS INSTITUTION

(Affiliated To University Of Mumbai)

Reaccredited 'A' Grade by NAAC | ISO 9001:2015 Certified

CERTIFICATE

Name: Mr. Ellison Povo

Roll No: 4055 Programme: BSc CS Semester: III

This is certified to be a bonafide record of practical works done by the above student in the college laboratory for the course **Data Structures (Course Code: 2032UISPR)** for the partial fulfilment of Third Semester of BSc IT during the academic year 2020-21.

The journal work is the original study work that has been duly approved in the year 2020-21 by the undersigned.

External Examiner

Mr. Gangashankar Singh
(Subject-In-Charge)

Date of Examination: (College Stamp)

Class: S.Y. B.Sc. CS Sem- III **Roll No: 4055**

Subject: Data Structures

INDEX

Sr No	Date	Topic	Sign
1	04/09/2020	Implement the following for Array: a) Write a program to store the elements in 1-D array and provide an option to perform the operations like searching, sorting, merging, reversing the elements. b) Write a program to perform the Matrix addition, Multiplication and Transpose Operation.	
2	11/09/2020	Implement Linked List. Include options for insertion, deletion and search of a number, reverse the list and concatenate two linked lists.	
3	18/09/2020	Implement the following for Stack: a) Perform Stack operations using Array implementation. b. b) Implement Tower of Hanoi. c) WAP to scan a polynomial using linked list and add two polynomials. d) WAP to calculate factorial and to compute the factors of a given no. (i) using recursion, (ii) using iteration	
4	25/09/2020	Perform Queues operations using Circular Array implementation.	
5	01/10/2020	Write a program to search an element from a list. Give user the option to perform Linear or Binary search.	
6	09/10/2020	WAP to sort a list of elements. Give user the option to perform sorting using Insertion sort, Bubble sort or Selection sort.	
7	16/10/2020	Implement the following for Hashing: a) Write a program to implement	

		the collision technique. b) Write a program to implement the concept of linear probing.	
8	23/10/2020	Write a program for inorder, postorder and preorder traversal of tree.	

Name : Ellison Povo 4055 roll no: 346

1 Implement the following for Array:

a:

Aim : Write a program to store the elements in 1-D array and provide an option to perform the operations like searching, sorting, merging, reversing the elements.

Theory

What is searching?
Searching is the process of finding a given value position in a list of values. It decides whether a search key is present in the data or not. It is the algorithmic process of finding a particular item in a collection of items.

What are some searching algorithms?
Linear Search :
A simple approach is to do a linear search, i.e

•Start from the leftmost element of arr[] and one by one compare x with each element of arr[] •If x matches with an element, return the index.

•If x doesn't match with any of elements, return -1.

Binary search:

Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

Sorting:

Sorting means arranging the elements of a list in a specific order.

There are many sorting algorithms like:

- Bubble sort
- Merge sort
- Selection Sort
- Insersion Sort
- Quick sort etc

Merging:

Merging means to join two list.

Reversing:

Reversing means the to arrang e the elements of the list in reverse order.

In the Code below one of the searchig and sorting techniuques have been applied.

Name : Ellison Povo 4055 roll no: 346

```python
class ArrayModification:

    def linear_search(self, lst, n):
        for i in range(len(lst)):
            if lst[i] == n:
                return f'Position :{i}'
        return -1

    def insertion_sort(self, lst):
        for i in range(len(lst)):

            index = lst[i]

            k = i - 1

            while k >= 0 and lst[k] > index:
                lst[k + 1] = lst[k]
                k -= 1

            lst[k + 1] = index

        return lst

    def merge(self, lst1, lst2):
        return ArrayModification.insertion_sort(lst1 + lst2)

    def reverse(self, lst):
        return lst[::-1]


lst = [2, 9, 1, 7, 3, 5, 2]
Armod = ArrayModification()
print(Armod.linear_search(lst, 3))
```

Run

Run: main

```
/root/PycharmProjects/pythonProject/venv/bin/python /root/PycharmProjects/pythonProject/main.py
Position :4

Process finished with exit code 0
```

B:

Write a program to perform the Matrix addition, Multiplication and Transpose Operation.

Theory

Matrix Addition: Matrix addition is the operation of adding two matrices by adding the corresponding entries together. The matrix can be added only when the number of rows and columns of the first matrix is equal to the number of rows and columns of the second matrix.

Matrix Multiplication: We can multiply two matrices if, and only if, the number of columns in the first matrix equals the number of rows in the second matrix. Otherwise, the product of two matrices is undefined.

Matrix Transpose: If A=[aij] be a matrix of order m x n, then the matrix obtained by interchanging the rows and columns of A is known as Transpose of matrix A. Transpose of matrix A is represented by AT.

Code and Output:

Name : Ellison Povo 4055 roll no: 346

```python
Mat1 = [[3, 4, -6],
        [12, 71, 24],
        [21, 3, 21]]

Mat2 = [[2, 16, -16],
        [1, 7, -3],
        [-1, 3, 3]]
Mat3 = [[0, 0, 0, ],
        [0, 0, 0, ],
        [0, 0, 0, ]]

# Matrix Addition
for i in range(len(Mat1)):
    for j in range(len(Mat2[0])):
        for k in range(len(Mat2)):
            Mat3[i][j] += Mat1[i][k] + Mat2[k][j]

print(Mat3)

# Matrix Multiplication

Mat3 = [[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]]

for i in range(len(Mat1)):
    for j in range(len(Mat2[0])):
        for k in range(len(Mat2)):
            Mat3[i][j] += Mat1[i][k] * Mat2[k][j]

print(Mat3)

# matrix transpose
for i in map(list, zip(*Mat1)):
    print(i)
```

Run

Run: main

```
/root/PycharmProjects/pythonProject/venv/bin/python /root/PycharmProjects/pythonProject/main.py
[[3, 27, -15], [109, 133, 91], [47, 71, 29]]
[[16, 58, -78, 0], [71, 761, -333, 0], [24, 420, -282, 0]]
[3, 12, 21]
[4, 71, 3]
[-6, 24, 21]

Process finished with exit code 0
```

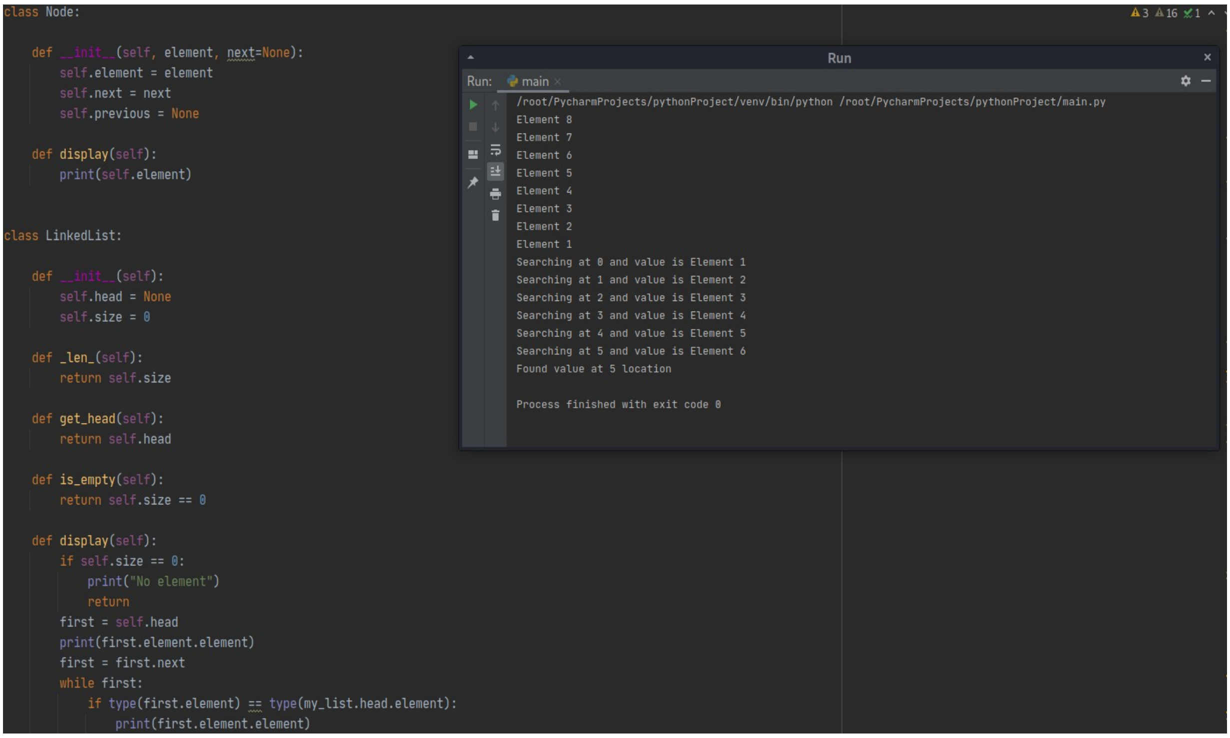
Name : Ellison Povo 4055 roll no: 346

2.

Aim: Implement Linked List. Include options for insertion, deletion and search of a number, reverse the list and concatenate two linked lists

Theory

Singly Linked List:
A singly linked list, in its simplest form, is a collection of nodes that collectively form a linear sequence. Each node stores a reference to an object that is an element of the sequence, as well as a reference to the next node of the list
Doubly Linked List:
In a singly linked list, each node maintains a reference to the node that is immediately after it. However, there are limitations that stem from the asymmetry of a singly linked list. To provide greater symmetry, we define a linked list in which each node keeps an explicit reference to the node before it and a reference to the node after it.
Such a structure is known as a doubly linked list. These lists allow a greater variety of O (1)-time update operations, including insertions and deletions at arbitrary positions within the list. We continue to use the term “next” for the reference to the node that follows another, and we introduce the term “prey” for the reference to the node that precedes it. With array-based sequences, an integer index was a convenient means for describing a position within a sequence. However, an index is not convenient for linked lists as there is no efficient way to find the jth element; it would seem to require a traversal of a portion of the list.
When working with a linked list, the most direct way to describe the location of an operation is by identifying a relevant node of the list. However, we prefer to encapsulate the inner workings of our data structure to avoid having users directly access nodes of a list.



```python
class Node:

    def __init__(self, element, next=None):
        self.element = element
        self.next = next
        self.previous = None

    def display(self):
        print(self.element)


class LinkedList:

    def __init__(self):
        self.head = None
        self.size = 0

    def _len_(self):
        return self.size

    def get_head(self):
        return self.head

    def is_empty(self):
        return self.size == 0

    def display(self):
        if self.size == 0:
            print("No element")
            return
        first = self.head
        print(first.element.element)
        first = first.next
        while first:
            if type(first.element) == type(my_list.head.element):
                print(first.element.element)
```

```
Run
Run: main
/root/PycharmProjects/pythonProject/venv/bin/python /root/PycharmProjects/pythonProject/main.py
Element 8
Element 7
Element 6
Element 5
Element 4
Element 3
Element 2
Element 1
Searching at 0 and value is Element 1
Searching at 1 and value is Element 2
Searching at 2 and value is Element 3
Searching at 3 and value is Element 4
Searching at 4 and value is Element 5
Searching at 5 and value is Element 6
Found value at 5 location

Process finished with exit code 0
```

Name : Ellison Povo 4055 roll no: 346

```python
def get_node_at(self, index):
    element_node = self.head
    counter = 0
    if index == 0:
        return element_node.element
    if index > self.size - 1:
        print("Index out of bound")
        return None
    while (counter < index):
        element_node = element_node.next
        counter += 1
    return element_node

def get_previous_node_at(self, index):
    if index == 0:
        print('No previous value')
        return None
    return my_list.get_node_at(index).previous

def remove_between_list(self, position):
    if position > self.size - 1:
        print("Index out of bound")
    elif position == self.size - 1:
        self.remove_tail()
    elif position == 0:
        self.remove_head()
    else:
        prev_node = self.get_node_at(position - 1)
        next_node = self.get_node_at(position + 1)
        prev_node.next = next_node
        next_node.previous = prev_node
        self.size -= 1

def add_between_list(self, position, element):
    element_node = Node(element)
```

Run

Run: main

```
/root/PycharmProjects/pythonProject/venv/bin/python /root/PycharmProjects/pythonProject/main.py
Element 8
Element 7
Element 6
Element 5
Element 4
Element 3
Element 2
Element 1
Searching at 0 and value is Element 1
Searching at 1 and value is Element 2
Searching at 2 and value is Element 3
Searching at 3 and value is Element 4
Searching at 4 and value is Element 5
Searching at 5 and value is Element 6
Found value at 5 location

Process finished with exit code 0
```

```python
    def add_tail(self, e):
        new_value = Node(e)
        new_value.previous = self.get_tail()
        self.get_tail().next = new_value
        self.size += 1

    def find_second_last_element(self):
        # second_last_element = None

        if self.size >= 2:
            first = self.head
            temp_counter = self.size - 2
            while temp_counter > 0:
                first = first.next
                temp_counter -= 1
            return first


        else:
            print("Size not sufficient")

        return None

    def remove_tail(self):
        if self.is_empty():
            print("Empty Singly linked list")
        elif self.size == 1:
            self.head == None
            self.size -= 1
        else:
            Node = self.find_second_last_element()
            if Node:
                Node.next = None
                self.size -= 1
```

Run

Run: main

```
/root/PycharmProjects/pythonProject/venv/bin/python /root/PycharmProjects/pythonProject/main.py
Element 8
Element 7
Element 6
Element 5
Element 4
Element 3
Element 2
Element 1
Searching at 0 and value is Element 1
Searching at 1 and value is Element 2
Searching at 2 and value is Element 3
Searching at 3 and value is Element 4
Searching at 4 and value is Element 5
Searching at 5 and value is Element 6
Found value at 5 location

Process finished with exit code 0
```

```python
def reverse_display(self):
    if self.size == 0:
        print("No element")
        return None
    last = my_list.get_tail()
    print(last.element)
    while last.previous:
        if type(last.previous.element) == type(my_list.head):
            print(last.previous.element.element)
            if last.previous == self.head:
                return None
            else:
                last = last.previous
        print(last.previous.element)
        last = last.previous

def add_head(self, e):
    # temp = self.head
    self.head = Node(e)
    # self.head.next = temp
    self.size += 1

def get_tail(self):
    last_object = self.head
    while (last_object.next != None):
        last_object = last_object.next
    return last_object

def remove_head(self):
    if self.is_empty():
        print("Empty Singly linked list")
    else:
        print("Removing")
        self.head = self.head.next
        self.head.previous = None
```

Run

Run: main

```
/root/PycharmProjects/pythonProject/venv/bin/python /root/PycharmProjects/pythonProject/main.py
Element 8
Element 7
Element 6
Element 5
Element 4
Element 3
Element 2
Element 1
Searching at 0 and value is Element 1
Searching at 1 and value is Element 2
Searching at 2 and value is Element 3
Searching at 3 and value is Element 4
Searching at 4 and value is Element 5
Searching at 5 and value is Element 6
Found value at 5 location

Process finished with exit code 0
```

Name : Ellison Povo 4055 roll no: 346

```python
def add_between_list(self, position, element):
    element_node = Node(element)
    if position > self.size:
        print("Index out of bound")
    elif position == self.size:
        self.add_tail(element)
    elif position == 0:
        self.add_head(element)
    else:
        prev_node = self.get_node_at(position - 1)
        current_node = self.get_node_at(position)
        prev_node.next = element_node
        element_node.previous = prev_node
        element_node.next = current_node
        current_node.previous = element_node
        self.size += 1

def search(self, search_value):
    index = 0
    while (index < self.size):
        value = self.get_node_at(index)
        if type(value.element) == type(my_list.head):
            print("Searching at " + str(index) + " and value is " + str(value.element.element))
        else:
            print("Searching at " + str(index) + " and value is " + str(value.element))
        if value.element == search_value:
            print("Found value at " + str(index) + " location")
            return True
        index += 1
    print("Not Found")
    return False

def merge(self, linkedlist_value):
    if self.size > 0:
        last_node = self.get_node_at(self.size - 1)
```

Run

Run: main

```
/root/PycharmProjects/pythonProject/venv/bin/python /root/PycharmProjects/pythonProject/main.py
Element 8
Element 7
Element 6
Element 5
Element 4
Element 3
Element 2
Element 1
Searching at 0 and value is Element 1
Searching at 1 and value is Element 2
Searching at 2 and value is Element 3
Searching at 3 and value is Element 4
Searching at 4 and value is Element 5
Searching at 5 and value is Element 6
Found value at 5 location

Process finished with exit code 0
```

```python
    if self.size > 0:
        last_node = self.get_node_at(self.size - 1)
        last_node.next = linkedlist_value.head
        linkedlist_value.head.previous = last_node
        self.size = self.size + linkedlist_value.size

    else:
        self.head = linkedlist_value.head
        self.size = linkedlist_value.size


l1 = Node('Element 1')
my_list = LinkedList()
my_list.add_head(l1)
my_list.add_tail('Element 2')
my_list.add_tail('Element 3')
my_list.add_tail('Element 4')
my_list.get_head().element.element
my_list.add_between_list(2, 'Element between')
my_list.remove_between_list(2)

my_list2 = LinkedList()
l2 = Node('Element 5')
my_list2.add_head(l2)
my_list2.add_tail('Element 6')
my_list2.add_tail('Element 7')
my_list2.add_tail('Element 8')
my_list.merge(my_list2)
my_list.get_previous_node_at(3).element
my_list.reverse_display()
my_list.search('Element 6')
```

Run

Run: main

```
/root/PycharmProjects/pythonProject/venv/bin/python /root/PycharmProjects/pythonProject/main.py
Element 8
Element 7
Element 6
Element 5
Element 4
Element 3
Element 2
Element 1
Searching at 0 and value is Element 1
Searching at 1 and value is Element 2
Searching at 2 and value is Element 3
Searching at 3 and value is Element 4
Searching at 4 and value is Element 5
Searching at 5 and value is Element 6
Found value at 5 location

Process finished with exit code 0
```

Name : Ellison Povo 4055 roll no: 346

3. Implement the following for Stack:

a.

Aim: Perform Stack operations using Array implementation.

Theory

Stack:

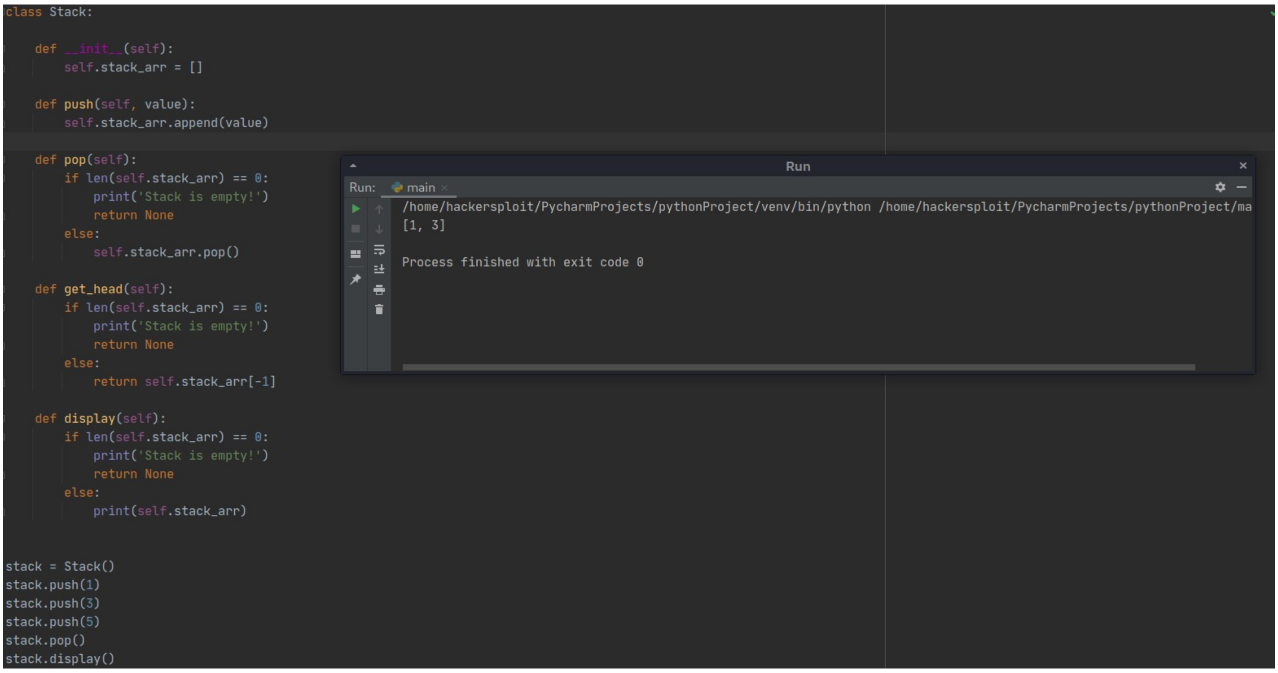
A stack is a collection of objects that are inserted and removed according to the last-in, first-out (LIFO) principle. A user may insert objects into a stack at any time, but may only access or remove the most recently inserted object that remains (at the so-called “top” of the stack). We can implement a stack quite easily by storing its elements in a Python list. The list class already supports adding an element to the end with the append method, and removing the last element with the pop method, so it is natural to align the top of the stack at the end of the list.

Stack is an abstract data type (ADT) such that an instance S supports the following two methods:

S.push(e): Add element e to the top of stack S.

S.pop(): Remove and return the top element from the stack S; an error occurs if the stack is empty.

Code and output:



```
class Stack:

    def __init__(self):
        self.stack_arr = []

    def push(self, value):
        self.stack_arr.append(value)

    def pop(self):
        if len(self.stack_arr) == 0:
            print('Stack is empty!')
            return None
        else:
            self.stack_arr.pop()

    def get_head(self):
        if len(self.stack_arr) == 0:
            print('Stack is empty!')
            return None
        else:
            return self.stack_arr[-1]

    def display(self):
        if len(self.stack_arr) == 0:
            print('Stack is empty!')
            return None
        else:
            print(self.stack_arr)


stack = Stack()
stack.push(1)
stack.push(3)
stack.push(5)
stack.pop()
stack.display()
```

Name : Ellison Povo 4055 roll no: 346

b.

Aim: Implement Tower of Hanoi

Theory:

Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:Only one disk can be moved at a time.

1. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
2. No disk may be placed on top of a smaller disk.

To write an algorithm for Tower of Hanoi, first we need to learn how to solve this problem with lesser number of disks, say → 1 or 2. We mark three towers with name, source, destination and aux (only to help moving the disks). If we have only one disk, then it can easily be moved from source to destination peg.

If we have 2 disks −

First, we move the smaller (top) disk to aux peg.

Then, we move the larger (bottom) disk to destination peg.

And finally, we move the smaller disk from aux to destination peg.

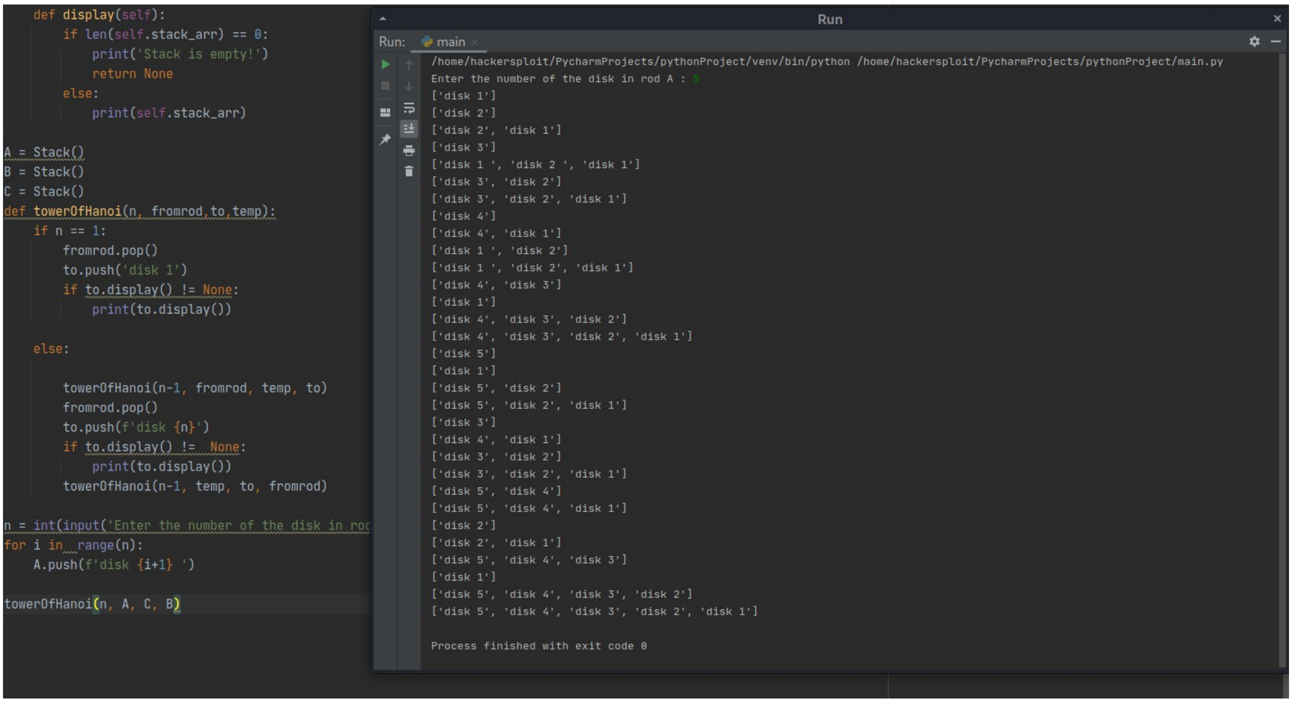
So now, we are in a position to design an algorithm for Tower of Hanoi with more than two disks. We divide the stack of disks in two parts. The largest disk (nth disk) is in one part and all other (n-1) disks are in the second part.

Our ultimate aim is to move disk n from source to destination and then put all other (n1) disks onto it. We can imagine to apply the same in a recursive way for all given set of disks. Each peg is a Stack object.

Code and output

```python
class Stack:

    def __init__(self):
        self.stack_arr = []

    def push(self,value):
        self.stack_arr.append(value)

    def pop(self):
        if len(self.stack_arr) == 0:
            print('Stack is empty!')
            return None
        else:
            self.stack_arr.pop()

    def get_head(self):
        if len(self.stack_arr) == 0:
            print('Stack is empty!')
            return None
        else:
            return self.stack_arr[-1]

    def display(self):
        if len(self.stack_arr) == 0:
            print('Stack is empty!')
            return None
        else:
            print(self.stack_arr)

A = Stack()
B = Stack()
C = Stack()
def towerOfHanoi(n, fromrod,to,temp):
    if n == 1:
        fromrod.pop()
        to.push('disk 1')
```

```
Run
Run: main
/home/hackersploit/PycharmProjects/pythonProject/venv/bin/python /home/hackersploit/PycharmProjects/pythonProject/main.py
Enter the number of the disk in rod A : 5
['disk 1']
['disk 2']
['disk 2', 'disk 1']
['disk 3']
['disk 1 ', 'disk 2 ', 'disk 1']
['disk 3', 'disk 2']
['disk 3', 'disk 2', 'disk 1']
['disk 4']
['disk 4', 'disk 1']
['disk 1 ', 'disk 2']
['disk 1 ', 'disk 2', 'disk 1']
['disk 4', 'disk 3']
['disk 1']
['disk 4', 'disk 3', 'disk 2']
['disk 4', 'disk 3', 'disk 2', 'disk 1']
['disk 5']
['disk 1']
['disk 5', 'disk 2']
['disk 5', 'disk 2', 'disk 1']
['disk 3']
['disk 4', 'disk 1']
['disk 3', 'disk 2']
['disk 3', 'disk 2', 'disk 1']
['disk 5', 'disk 4']
['disk 5', 'disk 4', 'disk 1']
['disk 2']
['disk 2', 'disk 1']
['disk 5', 'disk 4', 'disk 3']
['disk 1']
['disk 5', 'disk 4', 'disk 3', 'disk 2']
['disk 5', 'disk 4', 'disk 3', 'disk 2', 'disk 1']

Process finished with exit code 0
```



c.

Aim: WAP to scan a polynomial using linked list and add two polynomial.

Theory

Different operations can be performed on the polynomials like addition, subtraction, multiplication, and division. A polynomial is an expression within which a finite number of constants and variables are combined using addition, subtraction, multiplication, and exponents. Adding and subtracting polynomials is just adding and subtracting their like terms. The sum of two monomials is called a binomial and the sum of three monomials is called a trinomial. The sum of a finite number of monomials in x is called a polynomial in x. The coefficients of the monomials in a polynomial are called the coefficients of the polynomial. If all the coefficients of a polynomial are zero, then the polynomial is called the zero polynomial. Two polynomials can be added by using arithmetic operator plus (+). Adding polynomials is simply “combining like terms” and then add the like terms.
Every Polynomial in the program is a Doubly Linked List object. The corresponding terms are added and displayed in the form of an expression.

```python
class Node:

    def __init__(self, element, next=None):
        self.element = element
        self.next = next
        self.previous = None

    def display(self):
        print(self.element)


class LinkedList:

    def __init__(self):
        self.head = None
        self.size = 0

    def _len_(self):
        return self.size

    def get_head(self):
        return self.head

    def is_empty(self):
        return self.size == 0

    def display(self):
        if self.size == 0:
            print("No element")
            return
        first = self.head
        print(first.element.element)
        first = first.next
        while first:
            if type(first.element) == type(my_list.head.element):
                print(first.element.element)
```

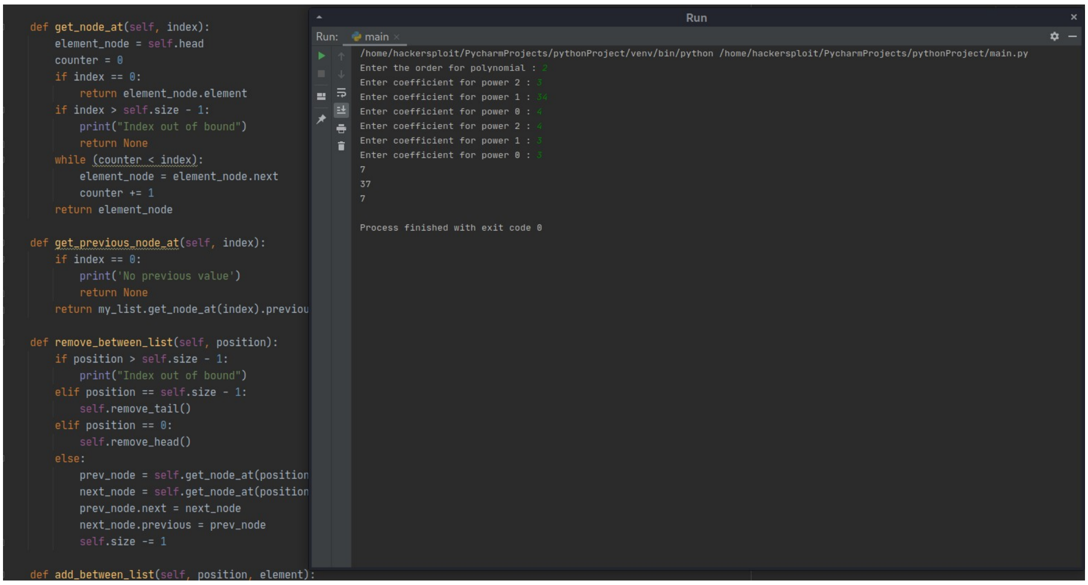
```
Run:  main
/home/hackersploit/PycharmProjects/pythonProject/venv/bin/python /home/hackersploit/PycharmProjects/pythonProject/main.py
Enter the order for polynomial : 2
Enter coefficient for power 2 : 3
Enter coefficient for power 1 : 4
Enter coefficient for power 0 : 3
Enter coefficient for power 2 : 1
Enter coefficient for power 1 : 2
Enter coefficient for power 0 : 3
4
6
6

Process finished with exit code 0
```

```python
    def reverse_display(self):
        if self.size == 0:
            print("No element")
            return None
        last = my_list.get_tail()
        print(last.element)
        while last.previous:
            if type(last.previous.element) == typ
                print(last.previous.element.eleme
                if last.previous == self.head:
                    return None
                else:
                    last = last.previous
            print(last.previous.element)
            last = last.previous

    def add_head(self, e):
        # temp = self.head
        self.head = Node(e)
        # self.head.next = temp
        self.size += 1

    def get_tail(self):
        last_object = self.head
        while (last_object.next != None):
            last_object = last_object.next
        return last_object

    def remove_head(self):
        if self.is_empty():
            print("Empty Singly linked list")
        else:
            print("Removing")
            self.head = self.head.next
            self.head.previous = None
```

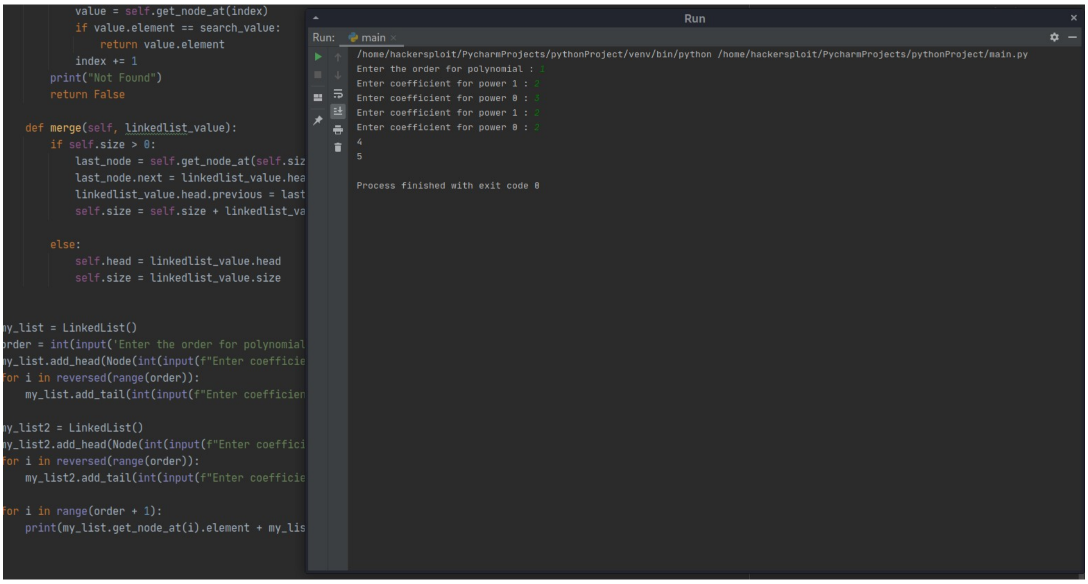
```
Run:  main
/home/hackersploit/PycharmProjects/pythonProject/venv/bin/python /home/hackersploit/PycharmProjects/pythonProject/main.py
Enter the order for polynomial : 2
Enter coefficient for power 2 : 3
Enter coefficient for power 1 : 4
Enter coefficient for power 0 : 3
Enter coefficient for power 2 : 1
Enter coefficient for power 1 : 2
Enter coefficient for power 0 : 3
4
6
6

Process finished with exit code 0
```

```python
            self.size -= 1

    def add_tail(self, e):
        new_value = Node(e)
        new_value.previous = self.get_tail()
        self.get_tail().next = new_value
        self.size += 1

    def find_second_last_element(self):
        # second_last_element = None

        if self.size >= 2:
            first = self.head
            temp_counter = self.size - 2
            while temp_counter > 0:
                first = first.next
                temp_counter -= 1
            return first


        else:
            print("Size not sufficient")

        return None

    def remove_tail(self):
        if self.is_empty():
            print("Empty Singly linked list")
        elif self.size == 1:
            self.head == None
            self.size -= 1
        else:
            Node = self.find_second_last_element(
            if Node:
                Node.next = None
```

```
Run:  main
/home/hackersploit/PycharmProjects/pythonProject/venv/bin/python /home/hackersploit/PycharmProjects/pythonProject/main.py
Enter the order for polynomial : 2
Enter coefficient for power 2 : 3
Enter coefficient for power 1 : 34
Enter coefficient for power 0 : 4
Enter coefficient for power 2 : 4
Enter coefficient for power 1 : 3
Enter coefficient for power 0 : 3
7
37
7

Process finished with exit code 0
```



def get_node_at(self, index):
element_node = self.head
counter = 0
if index == 0:
return element_node.element
if index > self.size - 1:
print("Index out of bound")
return None
while (counter < index):
element_node = element_node.next
counter += 1
return element_node
def get_previous_node_at(self, index):
if index == 0:
print('No previous value')
return None
return my_list.get_node_at(index).previou
def remove_between_list(self, position):
if position > self.size - 1:
print("Index out of bound")
elif position == self.size - 1:
self.remove_tail()
elif position == 0:
self.remove_head()
else:
prev_node = self.get_node_at(position
next_node = self.get_node_at(position
prev_node.next = next_node
next_node.previous = prev_node
self.size -= 1
def add_between_list(self, position, element):
Run
Run:
main
/home/hackersploit/PycharmProjects/pythonProject/venv/bin/python /home/hackersploit/PycharmProjects/pythonProject/main.py
Enter the order for polynomial : 2
Enter coefficient for power 2 : 3
Enter coefficient for power 1 : 34
Enter coefficient for power 0 : 4
Enter coefficient for power 2 : 4
Enter coefficient for power 1 : 3
Enter coefficient for power 0 : 3
7
37
7
Process finished with exit code 0



value = self.get_node_at(index)
if value.element == search_value:
return value.element
index += 1
print("Not Found")
return False
def merge(self, linkedlist_value):
if self.size > 0:
last_node = self.get_node_at(self.siz
last_node.next = linkedlist_value.hea
linkedlist_value.head.previous = last
self.size = self.size + linkedlist_va
else:
self.head = linkedlist_value.head
self.size = linkedlist_value.size
y_list = LinkedList()
rder = int(input('Enter the order for polynomial
y_list.add_head(Node(int(input(f"Enter coefficie
or i in reversed(range(order)):
my_list.add_tail(int(input(f"Enter coefficien
y_list2 = LinkedList()
y_list2.add_head(Node(int(input(f"Enter coeffici
or i in reversed(range(order)):
my_list2.add_tail(int(input(f"Enter coefficie
or i in range(order + 1):
print(my_list.get_node_at(i).element + my_lis
Run
Run:
main
/home/hackersploit/PycharmProjects/pythonProject/venv/bin/python /home/hackersploit/PycharmProjects/pythonProject/main.py
Enter the order for polynomial : 1
Enter coefficient for power 1 : 2
Enter coefficient for power 0 : 3
Enter coefficient for power 1 : 2
Enter coefficient for power 0 : 2
4
5
Process finished with exit code 0

d.

Aim: WAP to calculate factorial and to compute the factors of a given no. (i) using recursion, (ii) using iteration.

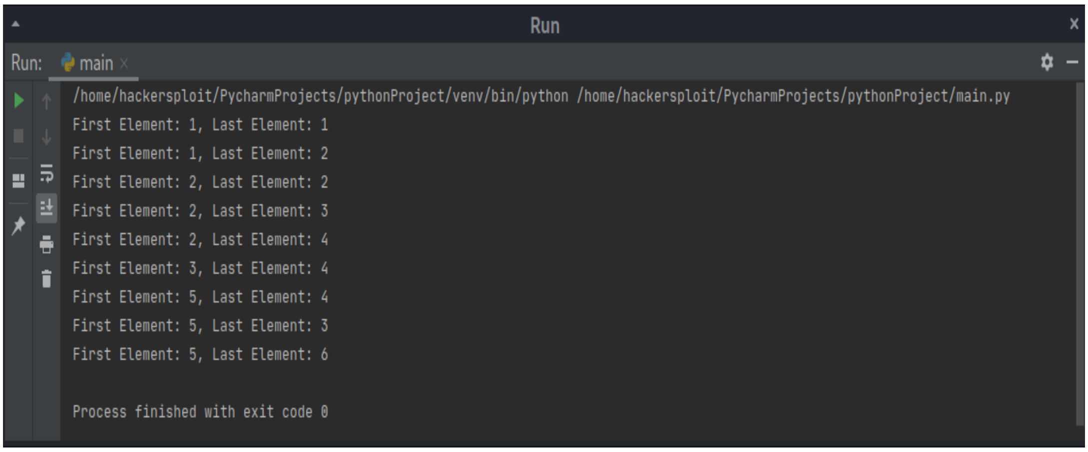
Theory

Factorial:
The factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 6 (denoted as 6!) is 1*2*3*4*5*6 = 720.
Factorial is not defined for negative numbers and the factorial of zero is one, 0! = 1. You can find it using recursion as well as iteration to calculate the factorial of a number. Factorial:
Factors are the numbers you multiply to get another number. For instance, factors of 15 are 3 and 5, because 3×5 = 15. Some numbers have more than one factorization (more than one way of being factored). For instance, 12 can be factored as 1×12, 2×6, or 3×4. A number that can only be factored as 1 time itself is called "prime".
You can find it using recursion as well as iteration to calculate the factors of a number.

4.
Aim: Perform Queues operations using Circular Array implementation.

Theory

Queue
the queue abstract data type defines a collection that keeps objects in a sequence, where element access and deletion are restricted to the first element in the queue, and element insertion is restricted to the back of the sequence. This restriction enforces the rule that items are inserted and deleted in a queue according to the first-in, first-out (FIFO) principle. The queue abstract data type (ADT) supports the following two fundamental methods for a queue Q: Q.enqueue(e): Add element e to the back of queue Q.
Q.dequeue( ): Remove and return the first element from queue Q; an error occurs if the queue is empty.
For the stack ADT, we created a very simple adapter class that used a Python list as the underlying storage.
Double Ended Queue
We next consider a queue-like data structure that supports insertion and deletion at both the front and the back of the queue. Such a structure is called a double ended queue, or deque, which is usually pronounced “deck” to avoid confusion with the dequeue method of the regular queue ADT, which is pronounced like the abbreviation “D.Q.”
The deque abstract data type is more general than both the stack and the queue ADTs.

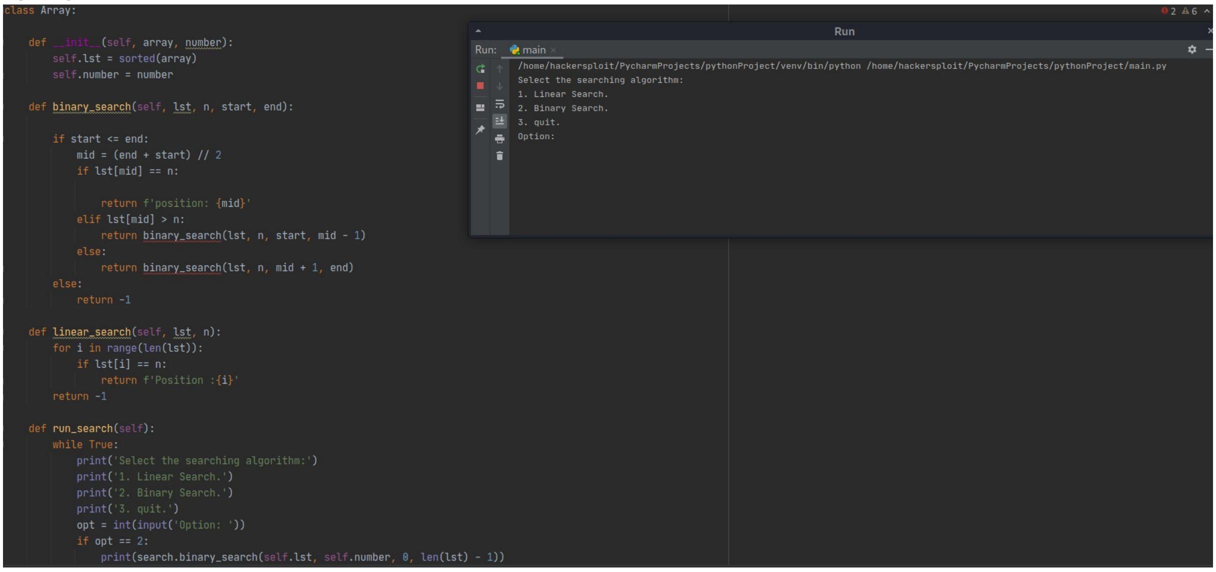


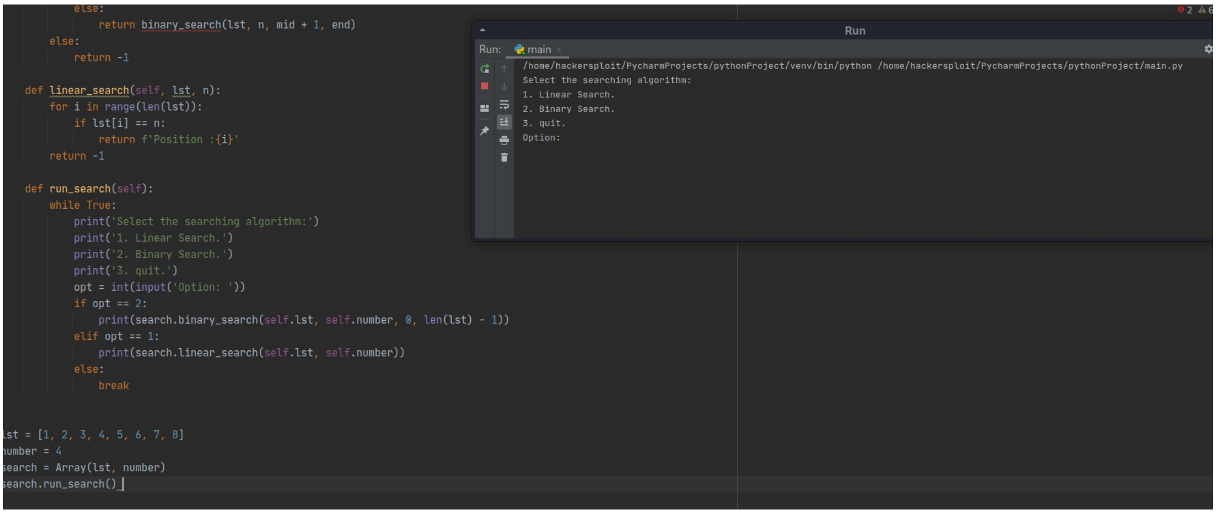
5.

Aim : Write a program to search an element from a list. Give user the option to perform
Linear or Binary search.

Theory

Binary Search: Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.
Linear Search: A Linear Search is the most basic type of searching algorithm. A Linear Search sequentially moves through your collection (or data structure) looking for a matching value. In other words, it looks down a list, one item at a time, without jumping.





6.
Aim: WAP to sort a list of elements. Give user the option to perform sorting using Insertion sort, Bubble sort or Selection sort.

Theory

Bubble sort : Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n2) where n is the number of items.

Insertion Sort : This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintainedto be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, insertion sort.

Selection Sort : Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

```
class Sorting:

    def __init__(self,lst):
        self.lst = lst

    def bubble_sort(self,lst):
        for i in range(len(lst)):
            for j in range(len(lst)):
                if lst[i] < lst[j]:
                    lst[i],lst[j] = lst[j],lst[i]
                else:
                    pass
        return lst

    def selection_sort(self,lst):
        for i in range(len(lst)):
            smallest_element = i
            for j in range(i+1,len(lst)):
                if lst[smallest_element] > lst[j]:
                    smallest_element = j
            lst[i],lst[smallest_element] = lst[smallest_element],lst[i]
        return lst

```

```python
    def insertion_sort(self,lst):
        for i in range(1, len(lst)):
            index = lst[i]
            j = i-1
            while j >= 0 and index < lst[j] :
                    lst[j + 1] = lst[j]
                    j -= 1
            lst[j + 1] = index
        return lst

    def run_sort(self):
        while True:
            print('Select the sorting algorithm:')
            print('1. Bubble Sort.')
            print('2. Selection Sort.')
            print('3. Insertion Sort.')
            print('4. Quit')
            opt = int(input('Option: '))
            if opt == 1:
                print(sort.bubble_sort(self.lst))
            elif opt == 2:
                print(sort.selection_sort(self.lst))
            elif opt == 3:
                print(sort.insertion_sort(self.lst))
            else:
                break
lst = [4,2,3,9,12,1]
sort = Sorting(lst)
sort.run_sort()
```

```
└──╼ #python practical_6.py
Select the sorting algorithm:
1. Bubble Sort.
2. Selection Sort.
3. Insertion Sort.
4. Quit
Option: 1
[1, 2, 3, 4, 9, 12]
Select the sorting algorithm:
1. Bubble Sort.
2. Selection Sort.
3. Insertion Sort.
4. Quit
Option: 2
[1, 2, 3, 4, 9, 12]
Select the sorting algorithm:
1. Bubble Sort.
2. Selection Sort.
3. Insertion Sort.
4. Quit
Option: 3
[1, 2, 3, 4, 9, 12]
Select the sorting algorithm:
1. Bubble Sort.
2. Selection Sort.
3. Insertion Sort.
4. Quit
Option: 4
┌─[root@parrot]─[/home/elliot/Desktop/Data-Structures-practicals]
└──╼ #
```

7. Implement the following for Hashing:

Aim: Write a program to implement the collision technique.

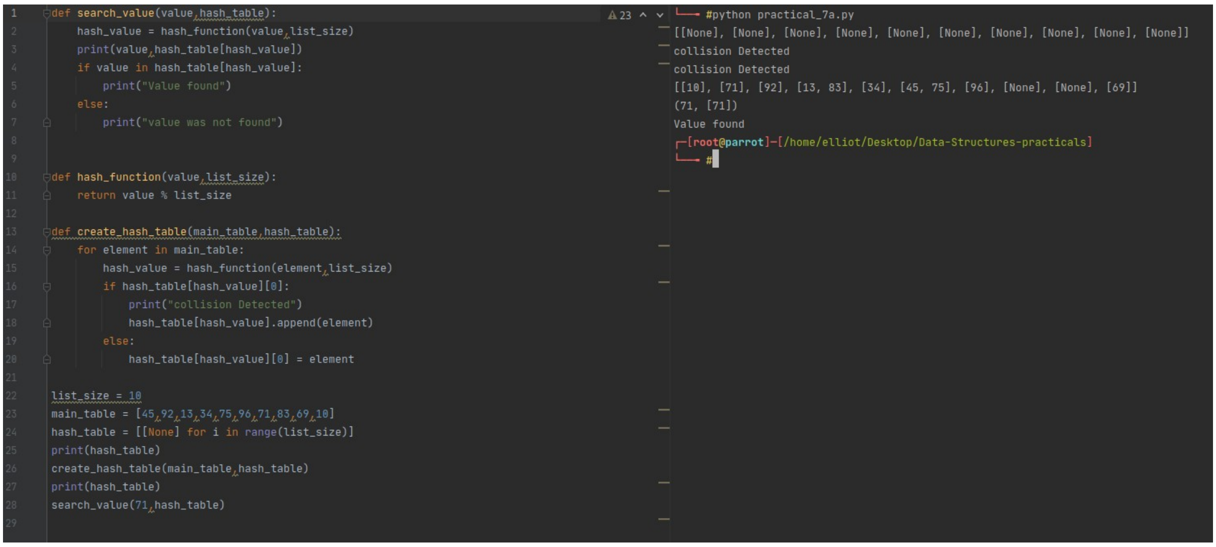
Theory

Hash Table is a data structure which stores data in an associative manner. In a hash table,
data is stored in an array format, where each data value has its own unique index value.
Access of data becomes very fast if we know the index of the desired data.
Thus, it becomes a data structure in which insertion and search operations are very fast
irrespective of the size of the data. Hash Table uses an array as a storage medium and
uses hash technique to generate an index where an element is to be inserted or is to be
located from.

Hashing

Hashing is a technique to convert a range of key values into a range of indexes of an
array. We're going to use modulo operator to get a range of key values. Consider an
example of hash table of size 20, and the following items are to be stored. Item are in the
(key,value) format.
In computer science, a collision or clash is a situation that occurs when two distinct pieces
of data have the same hash value, checksum, fingerprint, or cryptographic digest.[1]
Due to the possible applications of hash functions in data management and computer
security (in particular, cryptographic hash functions), collision avoidance has become a
fundamental topic in computer science.
Collisions are unavoidable whenever members of a very large set (such as all possible
person names, or all possible computer files) are mapped to a relatively short bit string.
This is merely an instance of the pigeonhole principle.
The impact of collisions depends on the application. When hash functions and fingerprints
are used to identify similar data, such as homologous DNA sequences or similar audio
files, the functions are designed so as to maximize the probability of collision between
distinct but similar data, using techniques like locality-sensitive hashing. Checksums, on

the other hand, are designed to minimize the probability of collisions between similar
inputs, without regard for collisions between very different inputs.



b.
Aim: Write a program to implement the concept of linear probing.
Theory
Linear probing is a scheme in computer programming for resolving collisions in hash tables, data structures for maintaining a collection of key–value pairs and looking up the value associated with a given key. It was invented in 1954 by Gene Amdahl, Elaine M. McGraw, and Arthur Samuel and first analyzed in 1963 by Donald Knuth.
Along with quadratic probing and double hashing, linear probing is a form of open addressing. In these schemes, each cell of a hash table stores a single key–value pair. When the hash function causes a collision by mapping a new key to a cell of the hash table that is already occupied by another key, linear probing searches the table for the closest following free location and inserts the new key there. Lookups are performed in the same way, by searching the table sequentially starting at the position given by the hash function, until finding a cell with a matching key or an empty cell

.

```python
def search_value(value, hash_table):
    hash_value = hash_function(value,list_size)
    if value in hash_table:
        print("Value found")
    else:
        print("value was not found")


def hash_function(value, list_size):
    return value % list_size


def create_hash_table( main_table , hash_table):

    for element in main_table:
        print(hash_table)
        index = 0
        hash_value = hash_function(element,list_size)
        while index < list_size:

            if hash_table[hash_value]:

                print("collision Detected")
                print("Moving to another slot -->")
                if hash_value == (list_size-1):
                    hash_value = 0
                    index += 1
                else:
                    hash_value += 1
                    index += 1
            else:
                hash_table[hash_value] = element
                break

list_size = 10
main_table = [45,92,13,34,75,96,71,83,69,10]
hash_table = [None for i in range(list_size)]
print(hash_table)
create_hash_table(main_table,hash_table)
print(hash_table)
search_value(71, hash_table)
```

Syncing...

```
└──╼ #python practical_7b.py
[None, None, None, None, None, None, None, None, None, None]
[None, None, None, None, None, None, None, None, None, None]
[None, None, None, None, None, 45, None, None, None, None]
[None, None, 92, None, None, 45, None, None, None, None]
[None, None, 92, 13, None, 45, None, None, None, None]
[None, None, 92, 13, 34, 45, None, None, None, None]
collision Detected
Moving to another slot -->
[None, None, 92, 13, 34, 45, 75, None, None, None]
collision Detected
Moving to another slot -->
[None, None, 92, 13, 34, 45, 75, 96, None, None]
[None, 71, 92, 13, 34, 45, 75, 96, None, None]
collision Detected
Moving to another slot -->
collision Detected
Moving to another slot -->
collision Detected
Moving to another slot -->
collision Detected
Moving to another slot -->
collision Detected
Moving to another slot -->
[None, 71, 92, 13, 34, 45, 75, 96, 83, None]
[None, 71, 92, 13, 34, 45, 75, 96, 83, 69]
[10, 71, 92, 13, 34, 45, 75, 96, 83, 69]
Value found
┌─[root@parrot]─[/home/elliot/Desktop/Data-Structures-practicals]
└──╼ #
```

8.
Aim :Write a program for inorder, postorder and preorder traversal of tree.
Theory
Pre-order (NLR)
Access the data part of the current node.
Traverse the left subtree by recursively calling the pre-order function.
Traverse the right subtree by recursively calling the pre-order function.
The pre-order traversal is a topologically sorted one, because a parent node is processed
before any of its child nodes is done.
In-order (LNR)
Traverse the left subtree by recursively calling the in-order function.
Access the data part of the current node.
Traverse the right subtree by recursively calling the in-order function.
In a binary search tree ordered such that in each node the key is greater than all keys in its
left subtree and less than all keys in its right subtree, in-order traversal retrieves the keys in
ascending sorted order.[4]
Post-order (LRN)
Traverse the left subtree by recursively calling the post-order function.
Traverse the right subtree by recursively calling the post-order function.
Access the data part of the current node.
The trace of a traversal is called a sequentialisation of the tree. The traversal trace is a list of
each visited root. No one sequentialisation according to pre-, in- or post-order describes the
underlying tree uniquely. Given a tree with distinct elements, either pre-order or post-order
paired with in-order is sufficient to describe the tree uniquely. However, pre-order with post-
order leaves some ambiguity in the tree structure.

```python
class Node:

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

    def PrintTree(self):
        if self.left:
            self.left.PrintTree()
        print(self.value)
        if self.right:
            self.right.PrintTree()

    def Printpreorder(self):
        if self.value:
            print(self.value)
            if self.left:
                self.left.Printpreorder()
            if self.right:
                self.right.Printpreorder()

    def Printinorder(self):
        if self.value:
            if self.left:
                self.left.Printinorder()
            print(self.value)
            if self.right:
                self.right.Printinorder()

    def Printpostorder(self):
        if self.value:
            if self.left:
                self.left.Printpostorder()
            if self.right:
                self.right.Printpostorder()
            print(self.value)
```

```python
    def insert(self, data):
        if self.value:
            if data < self.value:
                if self.left is None:
                    self.left = Node(data)
                else:
                    self.left.insert(data)
            elif data > self.value:
                if self.right is None:
                    self.right = Node(data)
                else:
                    self.right.insert(data)
        else:
            self.value = data

    def search_value(self,value):
        if self.left:
            self.left.search_value(value)
        if self.value == value:
            print("Value Found")
            return None
        if self.right:
            self.right.search_value(value)
        else:
            print("value not found")


root = Node(12)
root.insert(13)
root.insert(14)
root.insert(15)
root.insert(16)
root.insert(17)
root.PrintTree()
print("Pre order")
root.Printpreorder()
print("In Order")
root.Printinorder()
print("Post Order")
root.Printpostorder()
root.search_value(17)
```

```
└──╼ #python practical_8.py
12
13
14
15
16
17
Pre order
12
13
14
15
16
17
In Order
12
13
14
15
16
17
Post Order
17
16
15
14
13
12
Value Found
┌─[root@parrot]─[/home/elliot/Desktop/Data-Structures-practicals]
└──╼ #
```