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. Vivek Shivshankar Jayswal
Roll No: 316 Programme: BSc.IT 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. IT Sem- III **Roll No: 316**

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.	

Practical 1
Practical 1A. Implement the following for Array:
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:
Storing Data in Arrays. Assigning values to an element in an array is similar to assigning values to scalar variables. Simply reference an individual element of an array using the array name and the index inside parentheses, then use the assignment operator (=) followed by a value.

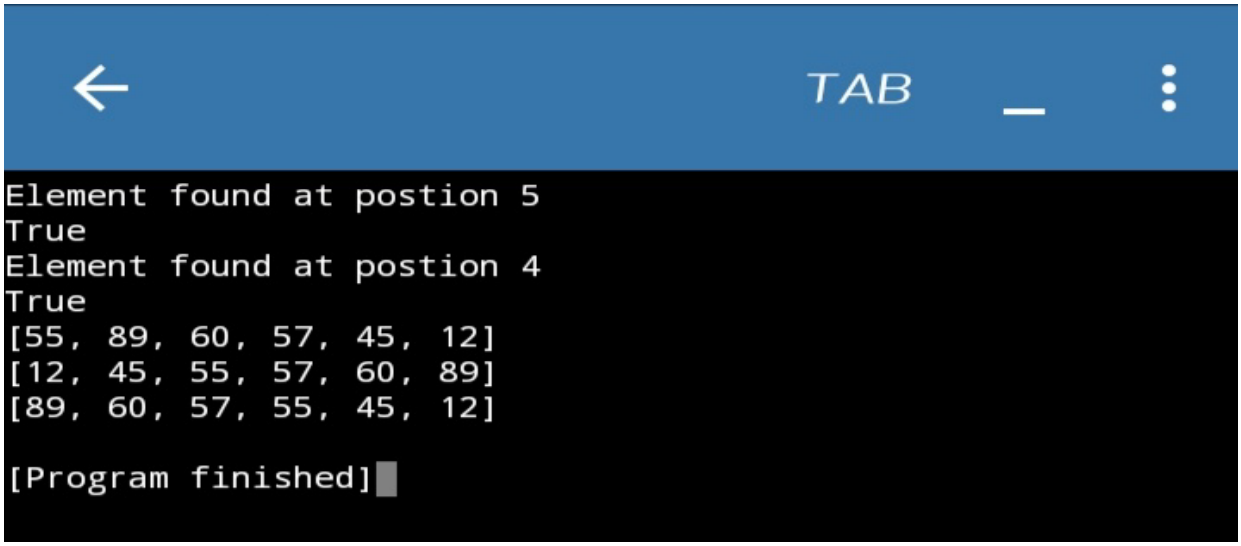
Following are the basic operations supported by an array.
Traverse − print all the array elements one by one.
Insertion − Adds an element at the given index.
Deletion − Deletes an element at the given index.
Search − Searches an element using the given index or by the value.

practical 1 a...
/storage/emul...

```
1  def linear_search(values,
   search_for):
2      search_at = 0
3      search_res = False
4      while search_at < len(values) and
   search_res is False:
5          if values[search_at] ==
   search_for:
6              search_res = True
7              print("Element found at
   postion",search_at)
8          else:
9              search_at = search_at + 1
10     return search_res
11 list = [12,45,57,60,89,55]
12 print(linear_search(list,55))
13 print(linear_search(list,89))
14 list.reverse()
15 print(list)
16 list.sort()
17 print(list)
18 list.sort(reverse = True)
19 print(list)
20
```

Tab | : | ; | ' | #

Output:



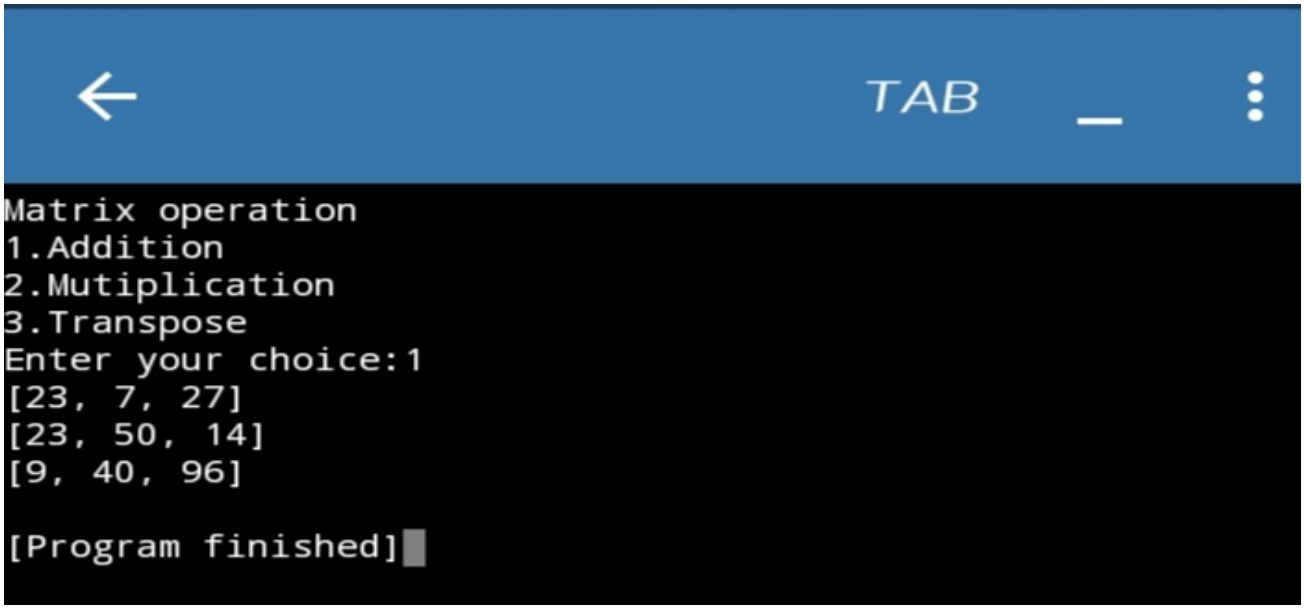
Practical 1B:
Aim: Write a program to perform the Matrix addition, Multiplication and Transpose Operation.

Theory: add() − add elements of two matrices.
subtract() − subtract elements of two matrices.
divide() − divide elements of two matrices.
multiply() − multiply elements of two matrices.
dot() − It performs matrix multiplication, does not element wise multiplication.
sqrt() − square root of each element of matrix.
sum(x,axis) − add to all the elements in matrix. Second argument is optional, it is used when we want to compute the column sum if axis is 0 and row sum if axis is 1.
“T” − It performs transpose of the specified matrix.

Pratical 1b.py
/storage/emul...

```
1  print("Matrix operation")
2  print("1.Addition")
3  print("2.Mutiplication")
4  print("3.Transpose")
5  x = [[12,3,21],[2,43,5],[5,32,53]]
6  y = [[11,4,6],[21,7,9],[4,8,43]]
7  result = [[0,0,0],[0,0,0],[0,0,0]]
8  ch = int(input("Enter your choice:"))
9  if ch == 1:
10     for i in range(len(x)):
11        for j in range(len(y[0])):
12           result[i][j] = x[i][j] + y[i][j]
13     for r in result:
14        print(r)
15 elif ch == 2:
16     for i in range(len(x)):
17        for j in range(len(y[0])):
18           for k in range (len(y)):
19              result[i][j] = x[i][j] * y[i][j]
20     for r in result:
21        print(r)
22 elif ch == 3:
23     for i in range(len(x)):
24        for j in range(len(y[0])):
25           result[j][i] = x[i][j]
26     for r in result:
27        print(r)
28 else:
29     print("Invalid choice")
30
```

Tab | : | ; | ' | # | (



Practical 2: Implement Linked List.

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

Theory: A linked list is a sequence of data elements, which are connected together via links. Each data element contains a connection to another data element in form of a pointer. Python does not have linked lists in its standard library. We implement the concept of linked lists using the concept of nodes as discussed in the previous chapter. We have already seen how we create a node class and how to traverse the elements of a node. In this chapter we are going to study the types of linked lists known as singly linked lists. In this type of data structure there is only one link between any two data elements. We create such a list and create additional methods to insert, update and remove elements from the list.

Insertion in a Linked List
Inserting element in the linked list involves reassigning the pointers from the existing nodes to the newly inserted node. Depending on whether the new data element is getting inserted at the beginning or at the middle or at the end of the linked list.

Deleting an Item form a Linked List
We can remove an existing node using the key for that node. In the below program we locate the previous node of the node which is to be deleted. Then point the next pointer of this node to the next node of the node to be deleted.

Searching in linked list

Searching is performed in order to find the location of a particular element in the list. Searching any element in the list needs traversing through the list and make the comparison of every element of the list with the specified element. If the element is matched with any of the list element then the location of the element is returned from the function.

Reversing a Linked List

To reverse a LinkedList recursively we need to divide the LinkedList into two parts: head and remaining. Head points to the first element initially. Remaining points to the next element from the head. We traverse theLinkedList recursively until the second last element.

Concatenating Linked Lists

Concatenate the two lists by traversing the first list until we reach it's a tail node and then point the next of the tail node to the head node of the second list. Store this concatenated list in the first list

practical 2....
/storage/emul...

```
1  class Node:
2
3      def __init__ (self, element, next =
   None ):
4          self.element = element
5          self.next = next
6          self.previous = None
7      def display(self):
8          print(self.element)
9
10 class LinkedList:
11
12     def __init__(self):
13         self.head = None
14         self.size = 0
15
16
17
18     def _len_(self):
19         return self.size
20
21     def get_head(self):
22         return self.head
23
24
25     def is_empty(self):
26         return self.size == 0
27
28     def display(self):
29         if self.size == 0:
```

Tab | : | ; | ' | # | (

practical 2....
/storage/emul...

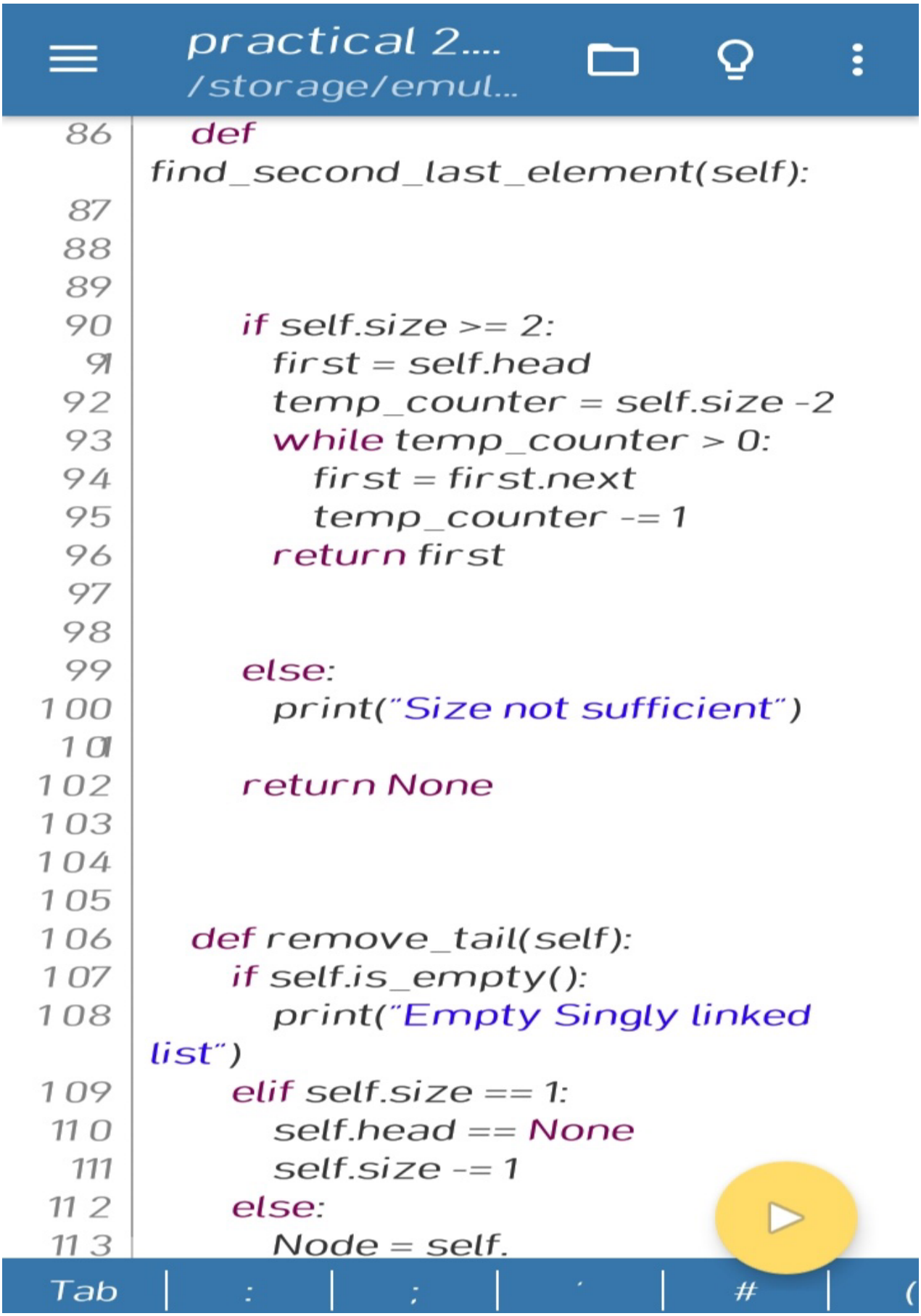
```
30          print("No element")
31          return
32        first = self.head
33        print(first.element.element)
34        first = first.next
35        while first:
36          if type(first.element) ==
type(list1.head.element):
37            print(first.element.element)
38            first = first.next
39          print(first.element)
40          first = first.next
41
42      def reverse_display(self):
43        if self.size == 0:
44          print("No element")
45          return None
46        last = list1.get_tail()
47        print(last.element)
48        while last.previous:
49          if type(last.previous.element)
== type(list1.head):
50            print(last.previous.element.
element)
51            if last.previous == self.head:
52              return None
53            else:
54              last = last.previous
55          print(last.previous.ele
56          last = last.previous
```

Tab | : | ; | ' | # |

practical 2....
/storage/emul...

```
59
60      def add_head(self,e):
61          self.head = Node(e)
62          self.size += 1
63
64      def get_tail(self):
65          last_object = self.head
66          while (last_object.next != None):
67              last_object = last_object.next
68          return last_object
69
70
71      def remove_head(self):
72          if self.is_empty():
73              print("Empty Singly linked
list")
74          else:
75              print("Removing")
76              self.head = self.head.next
77              self.head.previous = None
78              self.size -= 1
79
80      def add_tail(self,e):
81          new_value = Node(e)
82          new_value.previous = self.
get_tail()
83          self.get_tail().next = new_value
84          self.size += 1
85
86      def
```

Tab | : | ; | ' | # | (


practical 2....
/storage/emul...
86 def find_second_last_element(self):
87
88
89
90 if self.size >= 2:
91 first = self.head
92 temp_counter = self.size -2
93 while temp_counter > 0:
94 first = first.next
95 temp_counter -= 1
96 return first
97
98
99 else:
100 print("Size not sufficient")
101
102 return None
103
104
105
106 def remove_tail(self):
107 if self.is_empty():
108 print("Empty Singly linked list")
109 elif self.size == 1:
110 self.head == None
111 self.size -= 1
112 else:
113 Node = self.
Tab
:
;
'
#
(

practical 2....
/storage/emul...

```
113 find_second_last_element()
114         if Node:
115             Node.next = None
116             self.size -= 1
117
118     def get_node_at(self,index):
119         element_node = self.head
120         counter = 0
121         if index == 0:
122             return element_node.element
123         if index > self.size-1:
124             print("Index out of bound")
125             return None
126         while(counter < index):
127             element_node = element_node.
    next
128             counter += 1
129         return element_node
130
131     def get_previous_node_at(self,
    index):
132         if index == 0:
133             print('No previous value')
134             return None
135         return list1.get_node_at(index).
    previous
136
137     def remove_between_list(s
    position):
138         if position > self.size-1:
```

Tab | : | ; | ' | # | (

practical 2....
/storage/emul...

```
139             print("Index out of bound")
140         elif position == self.size-1:
141             self.remove_tail()
142         elif position == 0:
143             self.remove_head()
144         else:
145             prev_node = self.
    get_node_at(position-1)
146             next_node = self.
    get_node_at(position+1)
147             prev_node.next = next_node
148             next_node.previous =
    prev_node
149             self.size -= 1
150
151     def add_between_list(self,position,
    element):
152         element_node = Node(element)
153         if position > self.size:
154             print("Index out of bound")
155         elif position == self.size:
156             self.add_tail(element)
157         elif position == 0:
158             self.add_head(element)
159         else:
160             prev_node = self.
    get_node_at(position-1)
161             current_node = self.
    get_node_at(position)
162             prev_node.next =
```

Tab | : | ; | ' | # | (

practical 2....
/storage/emul...

```
162 element_node
163         element_node.previous =
    prev_node
164         element_node.next =
    current_node
165         current_node.previous =
    element_node
166         self.size += 1
167
168     def search (self,search_value):
169         index = 0
170         while (index < self.size):
171             value = self.get_node_at(index)
172             if type(value.element) ==
    type(list1.head):
173                 print("Searching at " +
    str(index) + " and value is " +
    str(value.element.element))
174             else:
175                 print("Searching at " +
    str(index) + " and value is " +
    str(value.element))
176             if value.element ==
    search_value:
177                 print("Found value at " +
    str(index) + " location")
178                 return True
179             index += 1
180         print("Not Found")
181         return False
```

Tab | : | ; | ' | # | (

practical 2....
/storage/emul...

```
182
183     def merge(self,linkedlist_value):
184         if self.size > 0:
185             last_node = self.
    get_node_at(self.size-1)
186             last_node.next =
    linkedlist_value.head
187             linkedlist_value.head.previous
    = last_node
188             self.size = self.size +
    linkedlist_value.size
189
190         else:
191             self.head = linkedlist_value.
    head
192             self.size = linkedlist_value.size
193
194 l1 = Node('element 1')
195 list1 = LinkedList()
196 list1.add_head(l1)
197 list1.add_tail('element 2')
198 list1.add_tail('element 3')
199 list1.add_tail('element 4')
200 list1.get_head().element.element
201 list1.add_between_list(2,'element
    between')
202 list1.remove_between_list(2)
203
204 list2 = LinkedList()
205 l2 = Node('element 5')
```

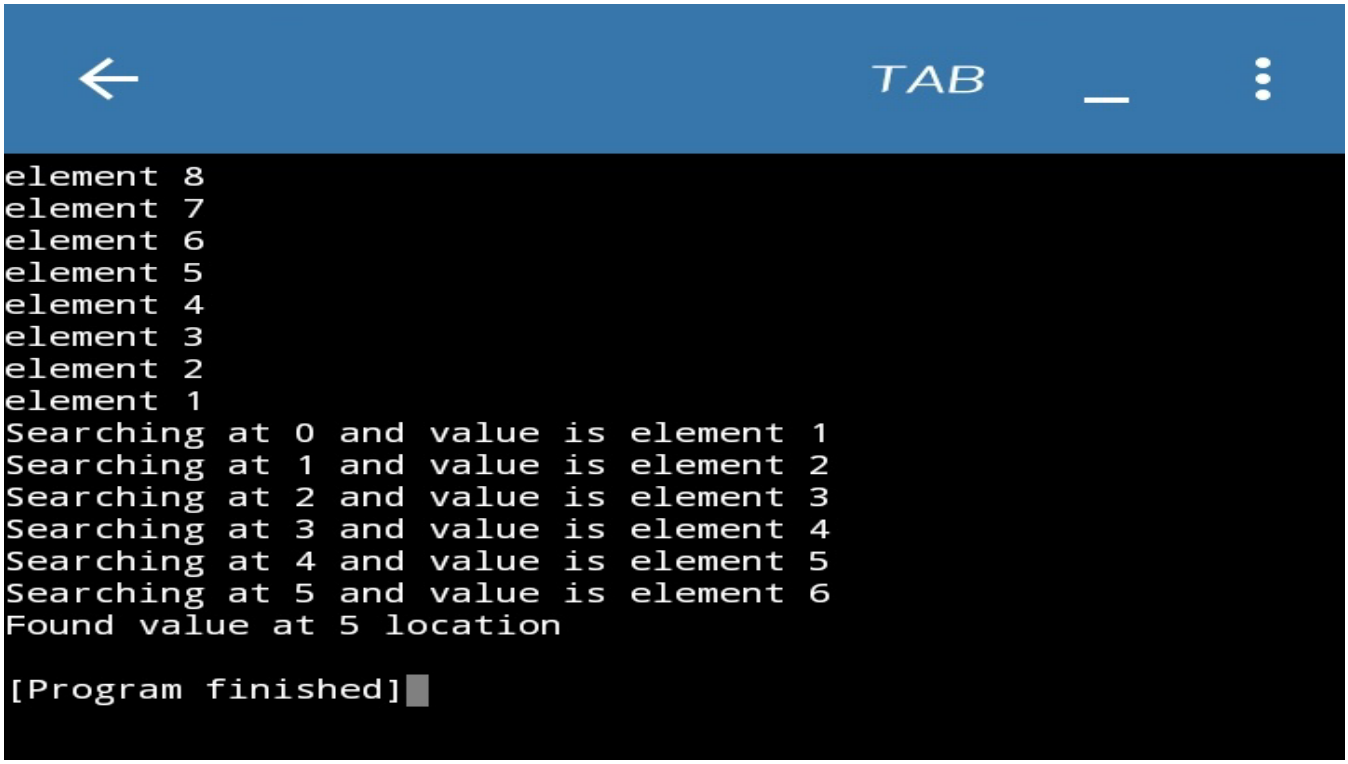
Tab | : | ; | ' | # | (

practical 2....
/storage/emul...

```
195 list1 = LinkedList()
196 list1.add_head(l1)
197 list1.add_tail('element 2')
198 list1.add_tail('element 3')
199 list1.add_tail('element 4')
200 list1.get_head().element.element
201 list1.add_between_list(2,'element
    between')
202 list1.remove_between_list(2)
203
204 list2 = LinkedList()
205 l2 = Node('element 5')
206 list2.add_head(l2)
207 list2.add_tail('element 6')
208 list2.add_tail('element 7')
209 list2.add_tail('element 8')
210 list1.merge(list2)
211 list1.get_previous_node_at(3).
    element
212 list1.reverse_display()
213 list1.search('element 6')
214
```

Tab | : | ; | ' | # | (

Output:



Practical 3. Implement the following for Stack:
Aim: Perform Stack operations using Array implementation.
Theory:
Stacks is one of the earliest data structures defined in computer science. In words, Stack is a linear collection of items. It is a collection of objects that supports fast last-in, first-out (LIFO) semantics for insertion and deletion. It is an array or list structure of function calls and parameters used in modern computer programming and CPU architecture. Similar to a stack of plates at a restaurant, elements in a stack are added or removed from the top of the stack, in a “last in, first out” order. Unlike lists or arrays, random access is not allowed for the objects contained in the stack.
There are two types of operations in Stack:

Push– To add data into the stack.
Pop– To remove data from the stack

Practical 3a...
/storage/emul...

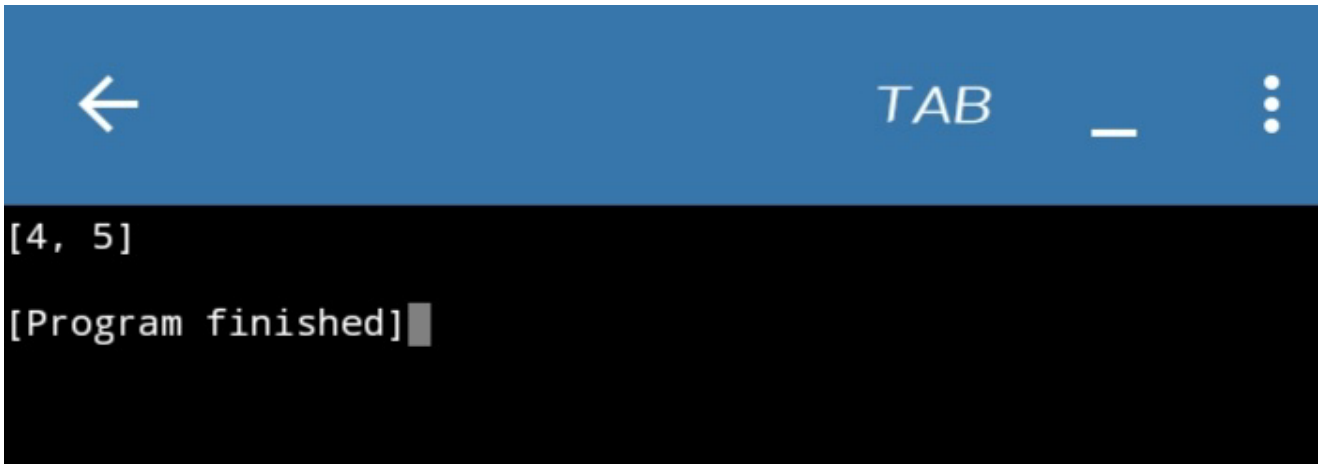
```
1  class Stack:
2
3      def __init__(self):
4          self.stack_arr = []
5
6      def push(self,value):
7          self.stack_arr.append(value)
8
9      def pop(self):
10         if len(self.stack_arr) == 0:
11             print('Stack is empty!')
12             return None
13         else:
14             self.stack_arr.pop()
15
16     def get_head(self):
17         if len(self.stack_arr) == 0:
18             print('Stack is empty!')
19             return None
20         else:
21             return self.stack_arr[-1]
22
23     def display(self):
24         if len(self.stack_arr) == 0:
25             print('Stack is empty!')
26             return None
27         else:
28             print(self.stack_arr)
29
30 stack = Stack()
```

Tab | : | ; | ' | # | (

Practical 3a...
/storage/emul...

```
15
16      def get_head(self):
17          if len(self.stack_arr) == 0:
18              print('Stack is empty!')
19              return None
20          else:
21              return self.stack_arr[-1]
22
23      def display(self):
24          if len(self.stack_arr) == 0:
25              print('Stack is empty!')
26              return None
27          else:
28              print(self.stack_arr)
29
30  stack = Stack()
31  stack.push(4)
32  stack.push(5)
33  stack.push(6)
34  stack.pop()
35  stack.display()
36  stack.get_head()
37
```

Tab | : | ; | ' | # |



Practical 3b:
Aim: b) Implement Tower of Hanoi.
Theory:
We are given n disks and a series of rods, we need to transfer all the disks to the final rod under the given constraints−
We can move only one disk at a time.
Only the uppermost disk

prac3b.py
/storage/emul...

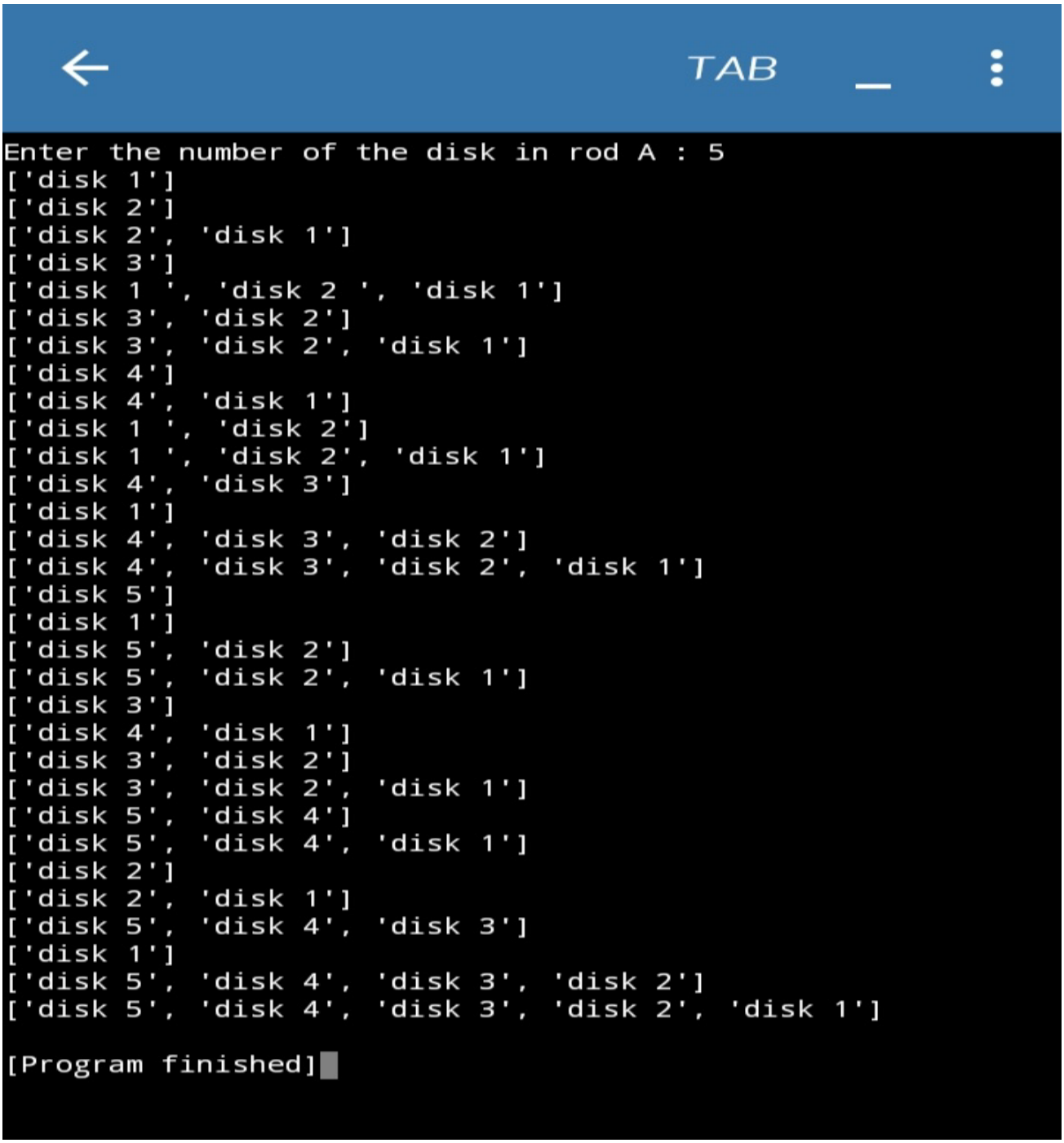
```
1  class Stack:
2
3      def __init__(self):
4          self.stack_arr = []
5
6      def push(self,value):
7          self.stack_arr.append(value)
8
9      def pop(self):
10         if len(self.stack_arr) == 0:
11             print('Stack is empty!')
12             return None
13         else:
14             self.stack_arr.pop()
15
16     def get_head(self):
17         if len(self.stack_arr) == 0:
18             print('Stack is empty!')
19             return None
20         else:
21             return self.stack_arr[-1]
22
23     def display(self):
24         if len(self.stack_arr) == 0:
25             print('Stack is empty!')
26             return None
27         else:
28             print(self.stack_arr)
29
30 A = Stack()
```

Tab | : | ; | ' | # | (

prac3b.py
/storage/emul...

```
B = Stack()
C = Stack()
def Hanoi(n, fromrod,to,temp):
    if n == 1:
        fromrod.pop()
        to.push('disk 1')
        if to.display() != None:
            print(to.display())

    else:

        Hanoi(n-1, fromrod, temp, to)
        fromrod.pop()
        to.push(f'disk {n}')
        if to.display() !=  None:
            print(to.display())
        Hanoi(n-1, temp, to, fromrod)

n = int(input('Enter the number of
the disk in rod A : '))
for i in  range(n):
    A.push(f'disk {i+1} ')

Hanoi(n, A, C, B)

```

Tab | : | ; | ' | # | (



Practical 3c:
Aim: WAP to scan a polynomial using linked list and add two polynomials.
Theory:
Polynomial is a mathematical expression that consists of variables and coefficients. for example x^2 - 4x +7
In the Polynomial linked list, the coefficients and exponents of the polynomial are defined as the data node of the list.

For adding two polynomials that are stored as a linked list. We need to add the coefficients of variables with the same power. In a linked list node contains 3 members, coefficient value link to the next node.
a linked list that is used to store Polynomial looks like −
Polynomial : 4x7 + 12x2 + 45

Practical 3c...
/storage/emul...

```
1  class Node:
2
3      def __init__ (self, element, next =
   None ):
4          self.element = element
5          self.next = next
6          self.previous = None
7      def display(self):
8          print(self.element)
9
10 class LinkedList:
11
12     def __init__(self):
13         self.head = None
14         self.size = 0
15
16
17
18     def _len_(self):
19         return self.size
20
21     def get_head(self):
22         return self.head
23
24
25     def is_empty(self):
26         return self.size == 0
27
28     def display(self):
29         if self.size == 0:
```

Tab | : | ; | ' | # | (

Practical 3c...
/storage/emul...

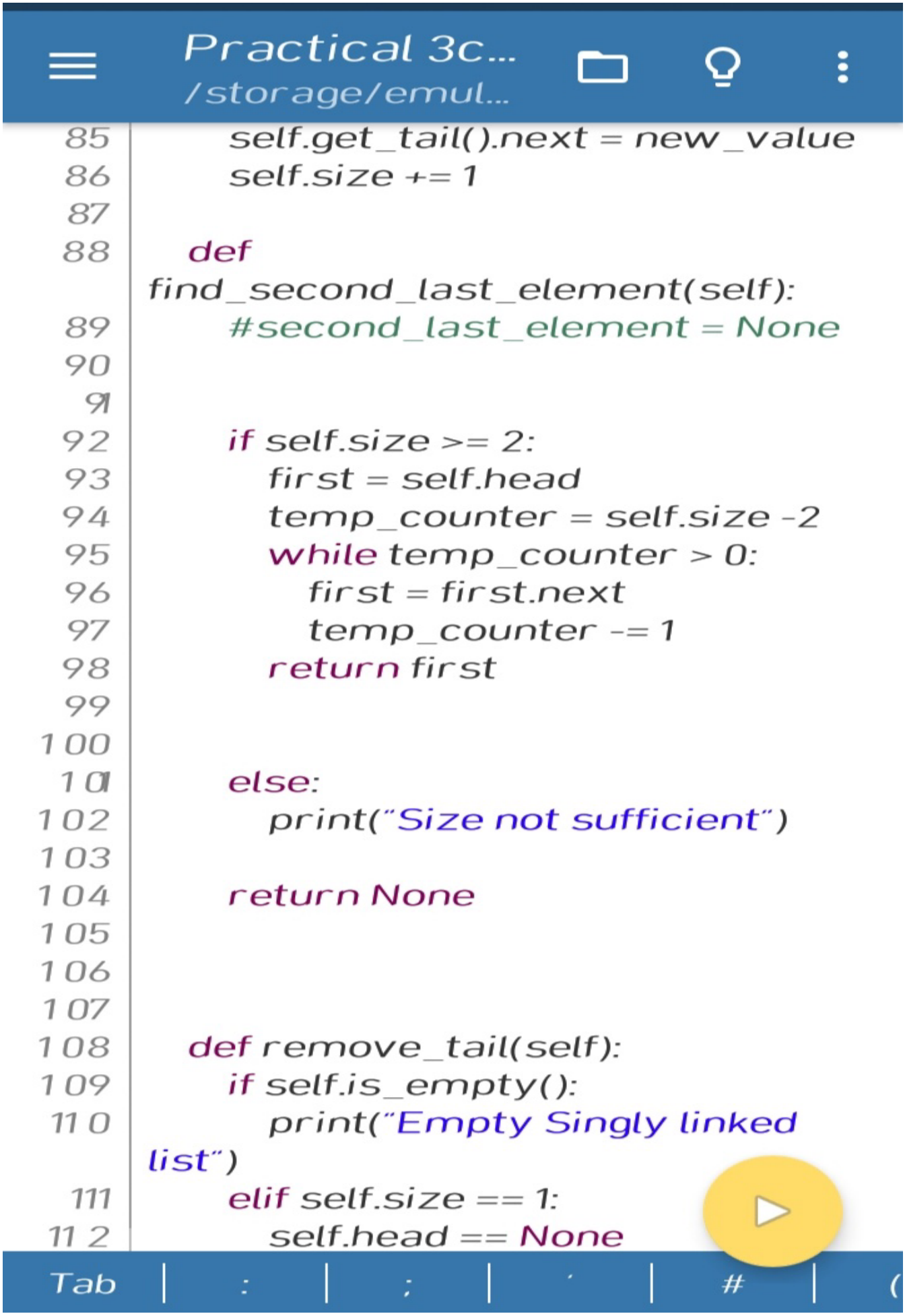
```
30          print("No element")
31          return
32      first = self.head
33      print(first.element.element)
34      first = first.next
35      while first:
36          if type(first.element) ==
    type(my_list.head.element):
37              print(first.element.element)
38              first = first.next
39          print(first.element)
40          first = first.next
41
42  def reverse_display(self):
43      if self.size == 0:
44          print("No element")
45          return None
46      last = my_list.get_tail()
47      print(last.element)
48      while last.previous:
49          if type(last.previous.element)
    == type(my_list.head):
50              print(last.previous.element.
    element)
51              if last.previous == self.head:
52                  return None
53              else:
54                  last = last.previous
55          print(last.previous.ele
56          last = last.previous
```

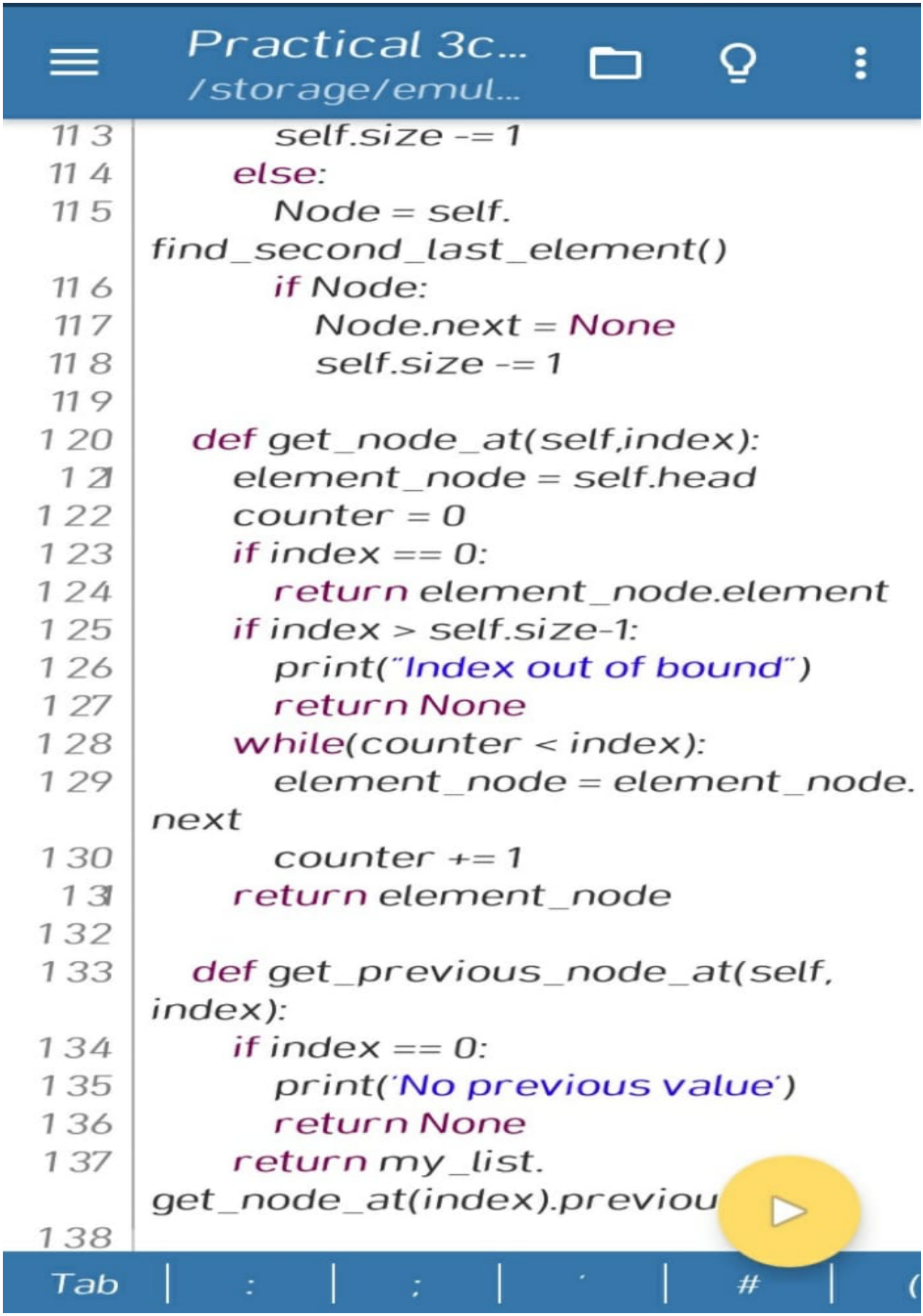
Tab | : | ; | ' | # | (

Practical 3c...
/storage/emul...

```
57
58
59
60      def add_head(self,e):
61          #temp = self.head
62          self.head = Node(e)
63          #self.head.next = temp
64          self.size += 1
65
66      def get_tail(self):
67          last_object = self.head
68          while (last_object.next != None):
69              last_object = last_object.next
70          return last_object
71
72
73      def remove_head(self):
74          if self.is_empty():
75              print("Empty Singly linked
list")
76          else:
77              print("Removing")
78              self.head = self.head.next
79              self.head.previous = None
80              self.size -= 1
81
82      def add_tail(self,e):
83          new_value = Node(e)
84          new_value.previous = s
get_tail()
```

Tab | : | ; | ' | # | (

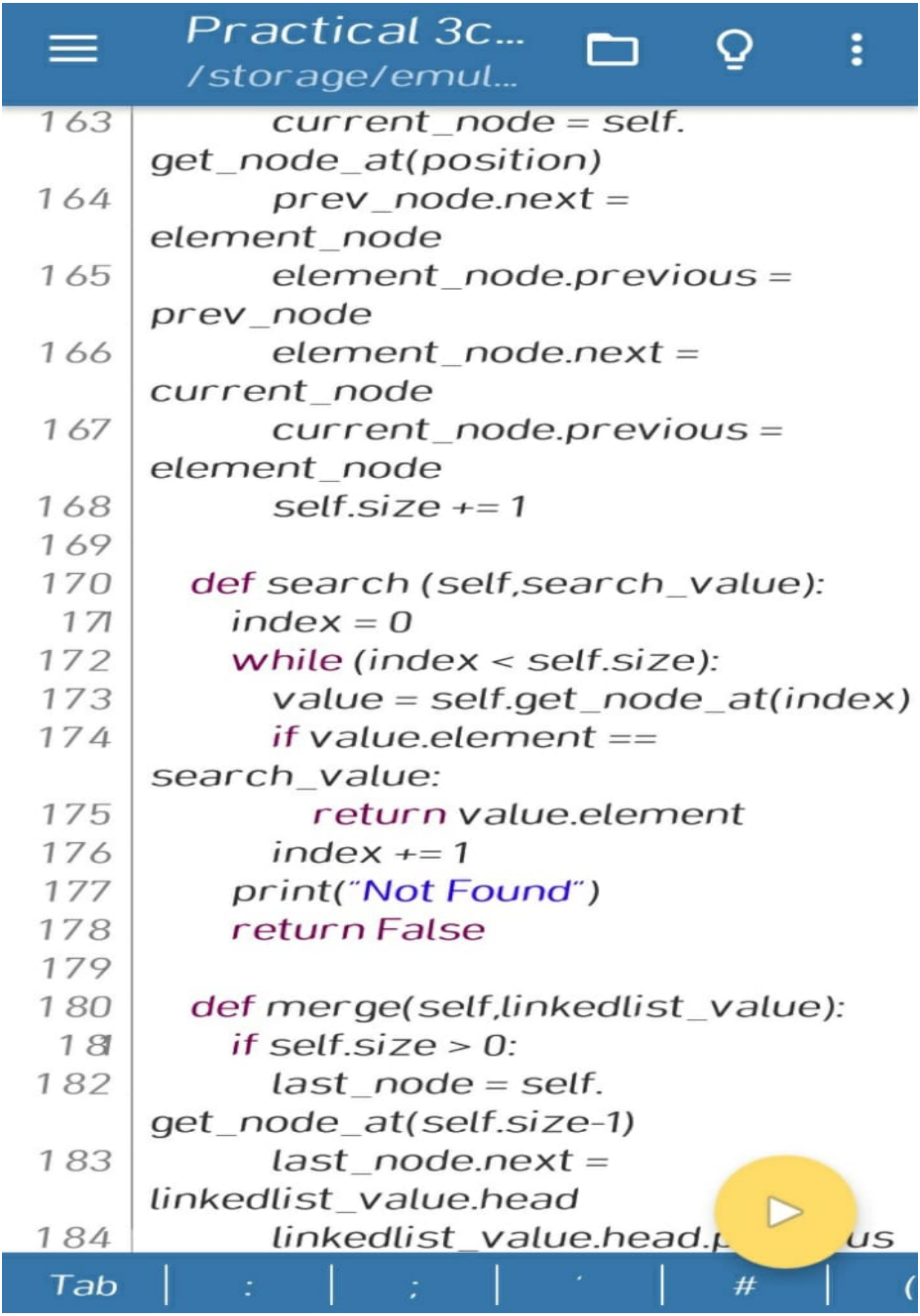

Practical 3c...
/storage/emul...
85 self.get_tail().next = new_value
86 self.size += 1
87
88 def find_second_last_element(self):
89 #second_last_element = None
90
91
92 if self.size >= 2:
93 first = self.head
94 temp_counter = self.size -2
95 while temp_counter > 0:
96 first = first.next
97 temp_counter -= 1
98 return first
99
100
101 else:
102 print("Size not sufficient")
103
104 return None
105
106
107
108 def remove_tail(self):
109 if self.is_empty():
110 print("Empty Singly linked list")
111 elif self.size == 1:
112 self.head == None
Tab : ; ' # (


Practical 3c...
/storage/emul...
113 self.size -= 1
114 else:
115 Node = self.
find_second_last_element()
116 if Node:
117 Node.next = None
118 self.size -= 1
119
120 def get_node_at(self,index):
121 element_node = self.head
122 counter = 0
123 if index == 0:
124 return element_node.element
125 if index > self.size-1:
126 print("Index out of bound")
127 return None
128 while(counter < index):
129 element_node = element_node.
next
130 counter += 1
131 return element_node
132
133 def get_previous_node_at(self,
index):
134 if index == 0:
135 print('No previous value')
136 return None
137 return my_list.
get_node_at(index).previou
138
Tab : ; ' # (

Practical 3c...
/storage/emul...

```
139     def remove_between_list(self, position):
140         if position > self.size-1:
141             print("Index out of bound")
142         elif position == self.size-1:
143             self.remove_tail()
144         elif position == 0:
145             self.remove_head()
146         else:
147             prev_node = self.get_node_at(position-1)
148             next_node = self.get_node_at(position+1)
149             prev_node.next = next_node
150             next_node.previous = prev_node
151             self.size -= 1
152
153     def add_between_list(self,position, element):
154         element_node = Node(element)
155         if position > self.size:
156             print("Index out of bound")
157         elif position == self.size:
158             self.add_tail(element)
159         elif position == 0:
160             self.add_head(element)
161         else:
162             prev_node = self.get_node_at(position-1)
```

Tab | : | ; | ' | # | (


Practical 3c...
/storage/emul...
163 current_node = self.
get_node_at(position)
164 prev_node.next =
element_node
165 element_node.previous =
prev_node
166 element_node.next =
current_node
167 current_node.previous =
element_node
168 self.size += 1
169
170 def search (self,search_value):
171 index = 0
172 while (index < self.size):
173 value = self.get_node_at(index)
174 if value.element ==
search_value:
175 return value.element
176 index += 1
177 print("Not Found")
178 return False
179
180 def merge(self,linkedlist_value):
181 if self.size > 0:
182 last_node = self.
get_node_at(self.size-1)
183 last_node.next =
linkedlist_value.head
184 linkedlist_value.head.p us
Tab : ; ' # (

Practical 3c...
/storage/emul...

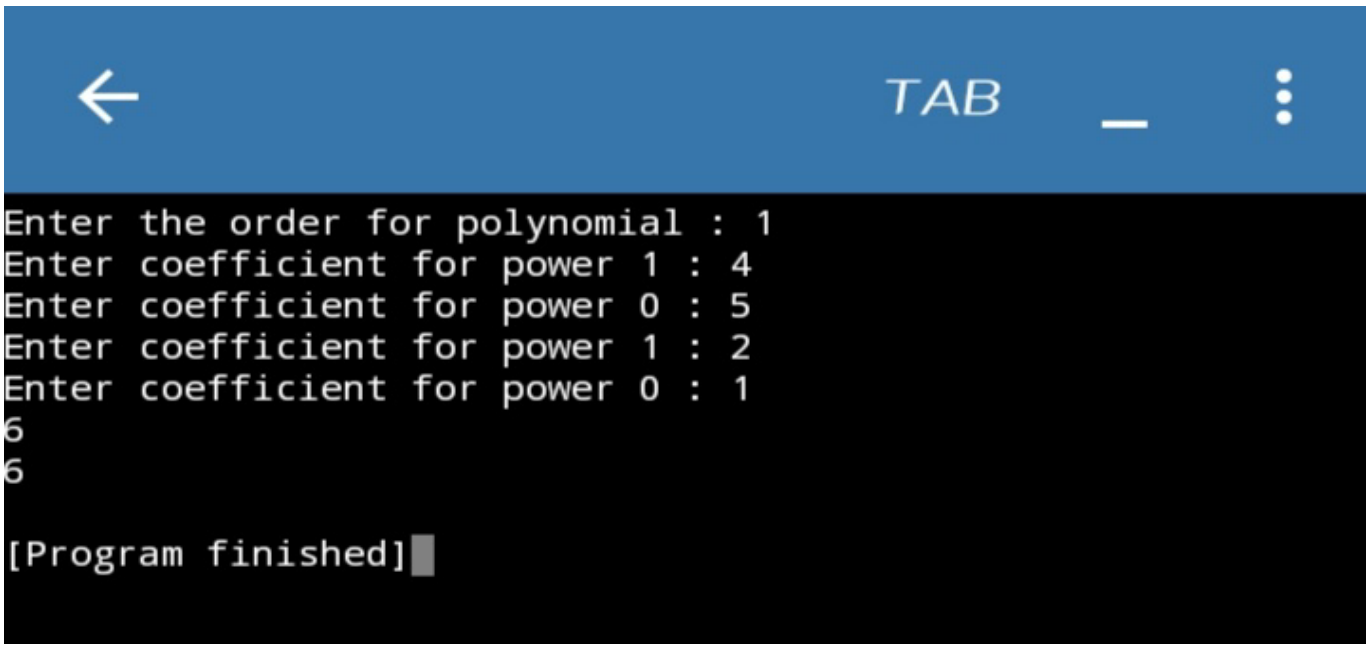
```
184 = last_node
185         self.size = self.size +
    linkedlist_value.size
186
187     else:
188         self.head = linkedlist_value.
    head
189         self.size = linkedlist_value.size
190
191
192
193 my_list = LinkedList()
194 order = int(input('Enter the order for
    polynomial : '))
195 my_list.
    add_head(Node(int(input(f"Enter
    coefficient for power {order} : "))))
196 for i in reversed(range(order)):
197     my_list.add_tail(int(input(f"Enter
    coefficient for power {i} : ")))
198
199 my_list2 = LinkedList()
200 my_list2.
    add_head(Node(int(input(f"Enter
    coefficient for power {order} : "))))
201 for i in reversed(range(order)):
202     my_list2.add_tail(int(input(f"Enter
    coefficient for power {i} : ")))
203
204 for i in range(order + 1):
```

Tab | : | ; | ' | # | (

Practical 3c...
/storage/emul...

```
193 my_list = LinkedList()
194 order = int(input('Enter the order for
    polynomial : '))
195 my_list.
    add_head(Node(int(input(f"Enter
    coefficient for power {order} : "))))
196 for i in reversed(range(order)):
197     my_list.add_tail(int(input(f"Enter
    coefficient for power {i} : ")))
198
199 my_list2 = LinkedList()
200 my_list2.
    add_head(Node(int(input(f"Enter
    coefficient for power {order} : "))))
201 for i in reversed(range(order)):
202     my_list2.add_tail(int(input(f"Enter
    coefficient for power {i} : ")))
203
204 for i in range(order + 1):
205     print(my_list.get_node_at(i).
    element + my_list2.get_node_at(i).
    element)
206
```

Tab | : | ; | ' | # | (



Practical 3d:
Aim: WAP to calculate factorial and to compute the factors of a given no.
(i) using recursion, (ii) using iteration
Theory:
The factorial of a number is the product of all the integers from 1 to that number. For example, the factorial of 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.

Recursion
In Python, we know that a function can call other functions. It is even possible for the function to call itself. These types of construct are termed as recursive functions.

Iteration
Repeating identical or similar tasks without making errors is something that computers do well and people do poorly. Repeated execution of a set of statements is called iteration. Because iteration is so common,Python provides several language features to make it easier.

Practical 3d...
/storage/emul...

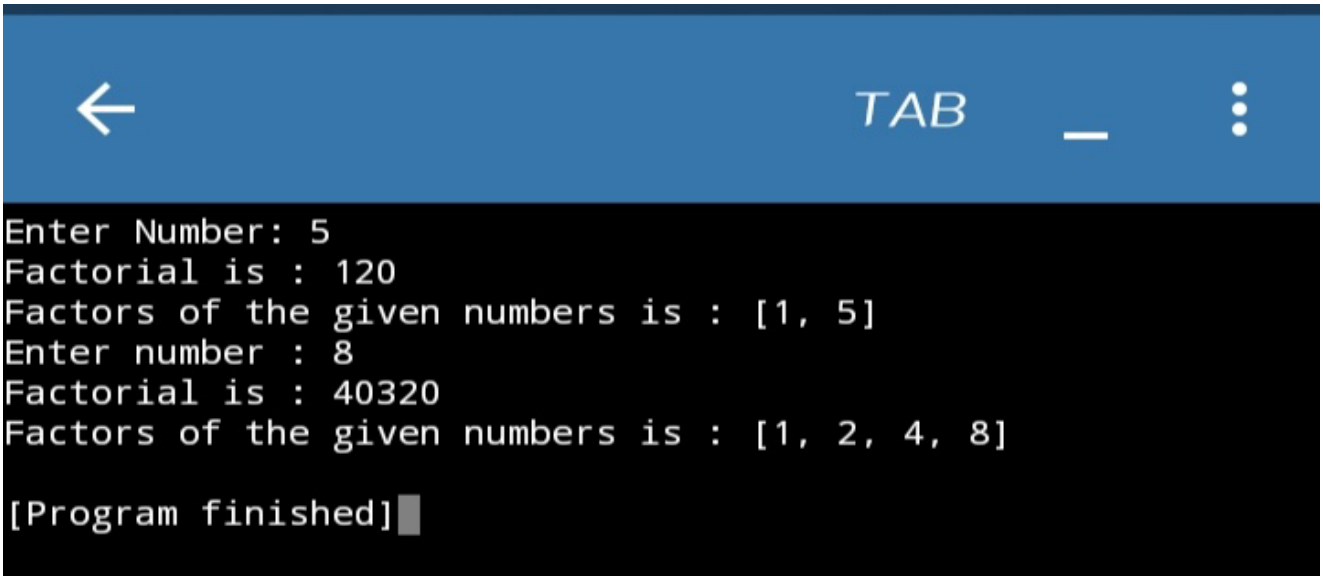
```
1  factorial = 1
2  n = int(input('Enter Number: '))
3  for i in range(1,n+1):
4      factorial = factorial * i
5
6  print(f'Factorial is : {factorial}')
7
8  fact = []
9  for i in range(1,n+1):
10     if (n/i).is_integer():
11         fact.append(i)
12
13 print(f'Factors of the given numbers
   is : {fact}')
14
15 factorial = 1
16 index = 1
17 n = int(input("Enter number : "))
18 def calculate_factorial(n,factorial,
   index):
19     if index == n:
20         print(f'Factorial is : {factorial}')
21         return True
22     else:
23         index = index + 1
24         calculate_factorial(n,factorial *
   index,index)
25 calculate_factorial(n,factorial,index)
26
27 fact = []
```

Tab | : | ; | ' | # | (

Practical 3d...
/storage/emul...

```
27  fact = []
28  def calculate_factors(n,factors,
    index):
29      if index == n+1:
30          print(f'Factors of the given
    numbers is : {factors}')
31          return True
32      elif (n/index).is_integer():
33          factors.append(index)
34          index += 1
35          calculate_factors(n,factors,
    index)
36      else:
37          index += 1
38          calculate_factors(n,factors,
    index)
39
40  index = 1
41  factors = []
42  calculate_factors(n,factors,index)
43
```

Tab | : | ; | ' | # | (



Practical 4:
Aim: Perform Queues operations using Circular Array implementation.
Theory: Circular queue avoids the wastage of space in a regular queue implementation using arrays.

Circular Queue works by the process of circular increment i.e. when we try to increment the pointer and we reach the end of the queue, we start from the beginning of the queue.
Here, the circular increment is performed by modulo division with the queue size. That is,
if REAR + 1 == 5 (overflow!), REAR = (REAR + 1)%5 = 0 (start of queue)

The circular queue work as follows:
two pointers FRONT and REAR

FRONT track the first element of the queue
REAR track the last elements of the queue

initially, set value of FRONT and rear to -1
1.Enqueue Operation

check if the queue is full
for the first element, set value of front to 0

circularly increase the REAR index by 1 (i.e. if the rear reaches the end, next it would be at the start of the queue)

add the new element in the position pointed to by REAR

2. Dequeue Operation

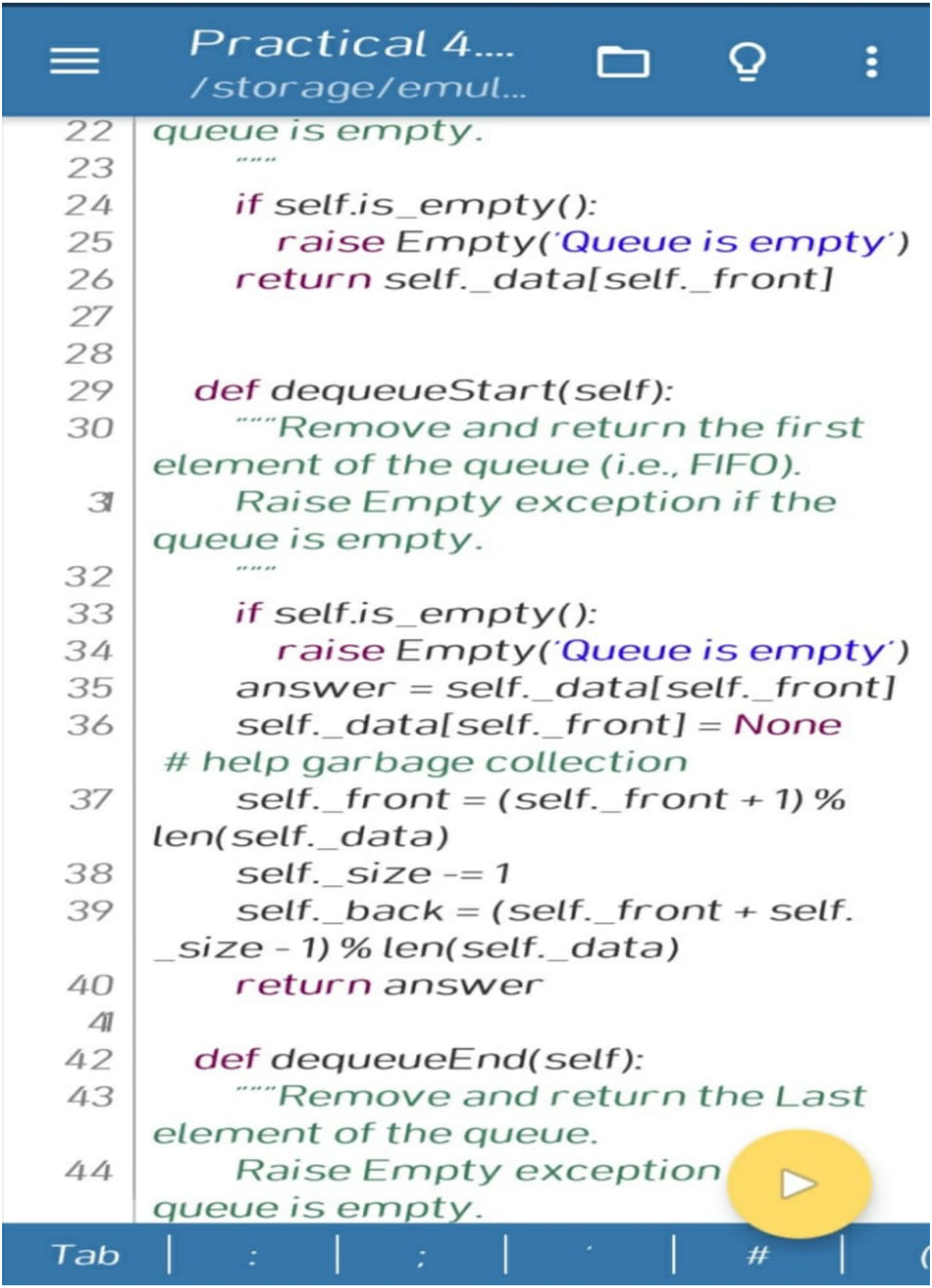
check if the queue is empty
return the value pointed by FRONT
circularly increase the FRONT index by 1

for the last element, reset the values of FRONT and REAR to -1

Practical 4....
/storage/emul...

```
1  class ArrayQueue:
2    """FIFO queue implementation
   using a Python list as underlying
   storage."""
3    DEFAULT_CAPACITY = 10        #
   moderate capacity for all new
   queues
4
5    def __init__(self):
6      """Create an empty queue."""
7      self._data = [None] * ArrayQueue.
   DEFAULT_CAPACITY
8      self._size = 0
9      self._front = 0
10     self._back = 0
11
12   def __len__(self):
13     """Return the number of
   elements in the queue."""
14     return self._size
15
16   def is_empty(self):
17     """Return True if the queue is
   empty."""
18     return self._size == 0
19
20   def first(self):
21     """Return (but do not rem
   the element at the front of t
22     Raise Empty exception
```

Tab | : | ; | ' | # | (


Practical 4....
/storage/emul...
queue is empty.
"""
if self.is_empty():
raise Empty('Queue is empty')
return self._data[self._front]
def dequeueStart(self):
"""Remove and return the first element of the queue (i.e., FIFO).
Raise Empty exception if the queue is empty.
"""
if self.is_empty():
raise Empty('Queue is empty')
answer = self._data[self._front]
self._data[self._front] = None # help garbage collection
self._front = (self._front + 1) % len(self._data)
self._size -= 1
self._back = (self._front + self._size - 1) % len(self._data)
return answer
def dequeueEnd(self):
"""Remove and return the Last element of the queue.
Raise Empty exception
queue is empty.
Tab : ; ' #

Practical 4....
/storage/emul...

```
45          """
46          if self.is_empty():
47              raise Empty('Queue is empty')
48          back = (self._front + self._size - 1)
    % len(self._data)
49          answer = self._data[back]
50          self._data[back] = None         #
    help garbage collection
51          self._front = self._front
52          self._size -= 1
53          self._back = (self._front + self.
    _size - 1) % len(self._data)
54          return answer
55
56      def enqueueEnd(self, e):
57          """Add an element to the back of
    queue."""
58          if self._size == len(self._data):
59              self._resize(2 * len(self.data))
     # double the array size
60          avail = (self._front + self._size) %
    len(self._data)
61          self._data[avail] = e
62          self._size += 1
63          self._back = (self._front + self.
    _size - 1) % len(self._data)
64
65      def enqueueStart(self, e):
66          """Add an element to the      of
    queue."""
```

Tab | : | ; | ' | # | (

Practical 4....
/storage/emul...

```
67          if self._size == len(self._data):
68              self._resize(2 * len(self._data))
              # double the array size
69          self._front = (self._front - 1) %
            len(self._data)
70          avail = (self._front + self._size) %
            len(self._data)
71          self._data[self._front] = e
72          self._size += 1
73          self._back = (self._front + self.
            _size - 1) % len(self._data)
74
75        def _resize(self, cap):                 #
            we assume cap >= len(self)
76          """Resize to a new list of
            capacity >= len(self)."""
77          old = self._data                    #
            keep track of existing list
78          self._data = [None] * cap           #
            allocate list with new capacity
79          walk = self._front
80          for k in range(self._size):         #
            only consider existing elements
81              self._data[k] = old[walk]
            # intentionally shift indices
82              walk = (1 + walk) % len(old)
            # use old size as modulus
83          self._front = 0                     #
            front has been realigned
84          self._back = (self._front
```

Tab | : | ; | ' | # | (

Practical 4....
/storage/emul...

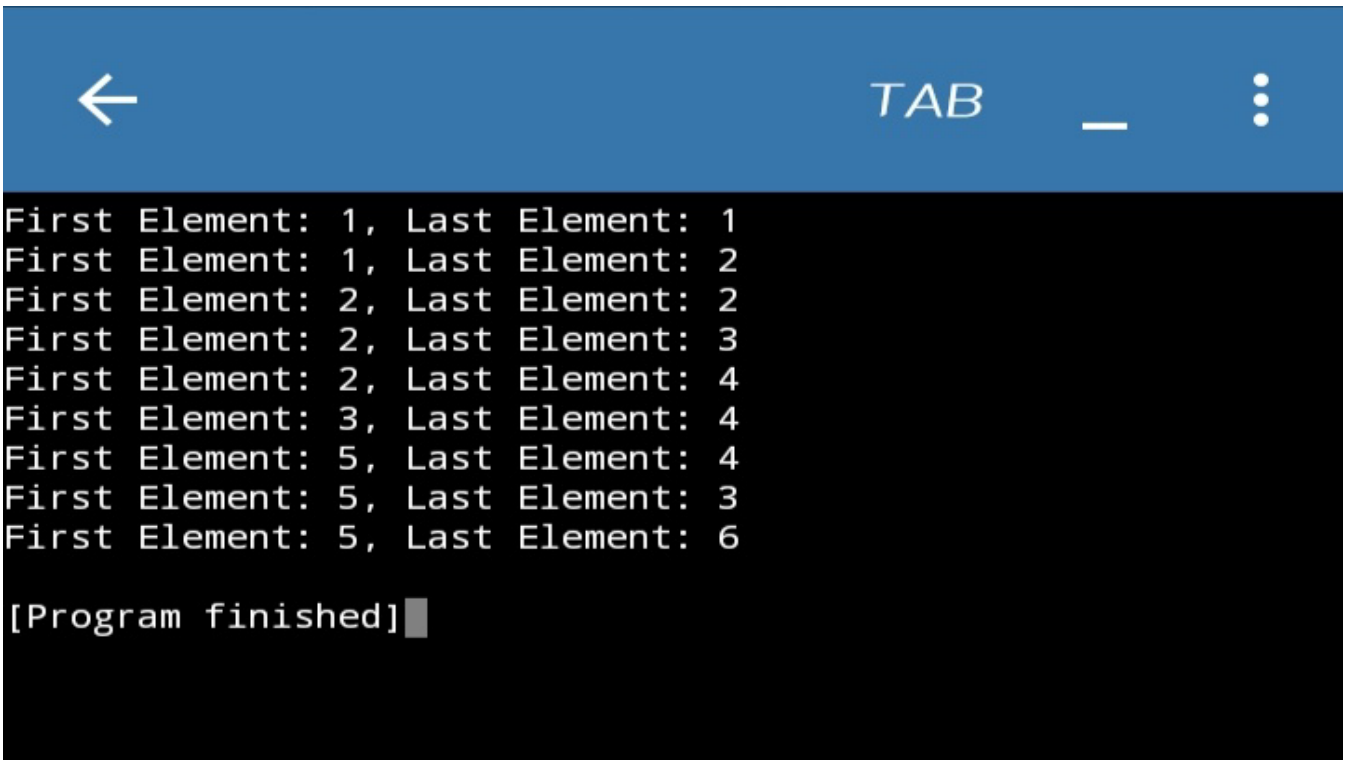
```
84  _size - 1) % len(self._data)
85
86  queue = ArrayQueue()
87  queue.enqueueEnd(1)
88  print(f"First Element: {queue.
    _data[queue._front]}, Last Element:
    {queue._data[queue._back]}")
89  queue._data
90  queue.enqueueEnd(2)
91  print(f"First Element: {queue.
    _data[queue._front]}, Last Element:
    {queue._data[queue._back]}")
92  queue._data
93  queue.dequeueStart()
94  print(f"First Element: {queue.
    _data[queue._front]}, Last Element:
    {queue._data[queue._back]}")
95  queue.enqueueEnd(3)
96  print(f"First Element: {queue.
    _data[queue._front]}, Last Element:
    {queue._data[queue._back]}")
97  queue.enqueueEnd(4)
98  print(f"First Element: {queue.
    _data[queue._front]}, Last Element:
    {queue._data[queue._back]}")
99  queue.dequeueStart()
100 print(f"First Element: {queue.
    _data[queue._front]}, Last Element:
    {queue._data[queue._back]}
101 queue.enqueueStart(5)
```

Tab | : | ; | ' | # |

Practical 4....
/storage/emul...

```
97  queue.enqueueEnd(4)
98  print(f"First Element: {queue.
    _data[queue._front]}, Last Element:
    {queue._data[queue._back]}")
99  queue.dequeueStart()
100 print(f"First Element: {queue.
    _data[queue._front]}, Last Element:
    {queue._data[queue._back]}")
101 queue.enqueueStart(5)
102 print(f"First Element: {queue.
    _data[queue._front]}, Last Element:
    {queue._data[queue._back]}")
103 queue.dequeueEnd()
104 print(f"First Element: {queue.
    _data[queue._front]}, Last Element:
    {queue._data[queue._back]}")
105 queue.enqueueEnd(6)
106 print(f"First Element: {queue.
    _data[queue._front]}, Last Element:
    {queue._data[queue._back]}")
107
```

Tab | : | ; | ' | # | (



Practical 5:
Aim: Write a program to search an element from a list. Give user the option to perform Linear or Binary search.
Theory:
Linear Search:
This linear search is a basic search algorithm which searches all the elements in the list and finds the required value. ... This is also known as sequential search.

Binary Search:
In computer science, a binary search or half-interval search algorithm finds the position of a target value within a sorted array. The binary search algorithm can be classified as a dichotomies divide-and-conquer search algorithm and executes in logarithmic time.

practical_5....
/storage/emul...

```
1  class Array:
2
3      def __init__(self,array,number):
4          self.lst = sorted(array)
5          self.number = number
6      def binary_search(self,lst,n,start,
   end):
7
8          if start <= end:
9              mid = (end + start) // 2
10             if lst[mid] == n:
11
12                 return f'position: {mid}'
13             elif lst[mid] > n:
14                 return binary_search(lst,n,
   start,mid-1)
15             else:
16                 return binary_search(lst,n,
   mid + 1,end)
17         else:
18             return -1
19
20
21     def linear_search(self,lst,n):
22         for i in range(len(lst)):
23             if lst[i] == n:
24                 return f'Position :{i}'
25         return -1
26
27     def run_search(self):
```

Tab | : | ; | ' | # | (

practical_5....
/storage/emul...

```
28          while True:
29              print('Select the searching
algorithm:')
30              print('1. Linear Search.')
31              print('2. Binary Search.')
32              print('3. quit.')
33              opt = int(input('Option: '))
34              if opt == 2:
35                  print(search.
binary_search(self.lst,self.number,0,
len(lst)-1))
36              elif opt == 1:
37                  print(search.
linear_search(self.lst,self.number))
38              else:
39                  break
40
41
42  lst = [1,2,3,4,5,6,7,8]
43  number = 4
44  search = Array(lst,number)
45  search.run_search()
46
47
48
```

Tab | : | ; | ' | # | (

Practical 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 the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

Selection Sort:
The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array

Insertion Sort:
Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

practical_6....
/storage/emul...

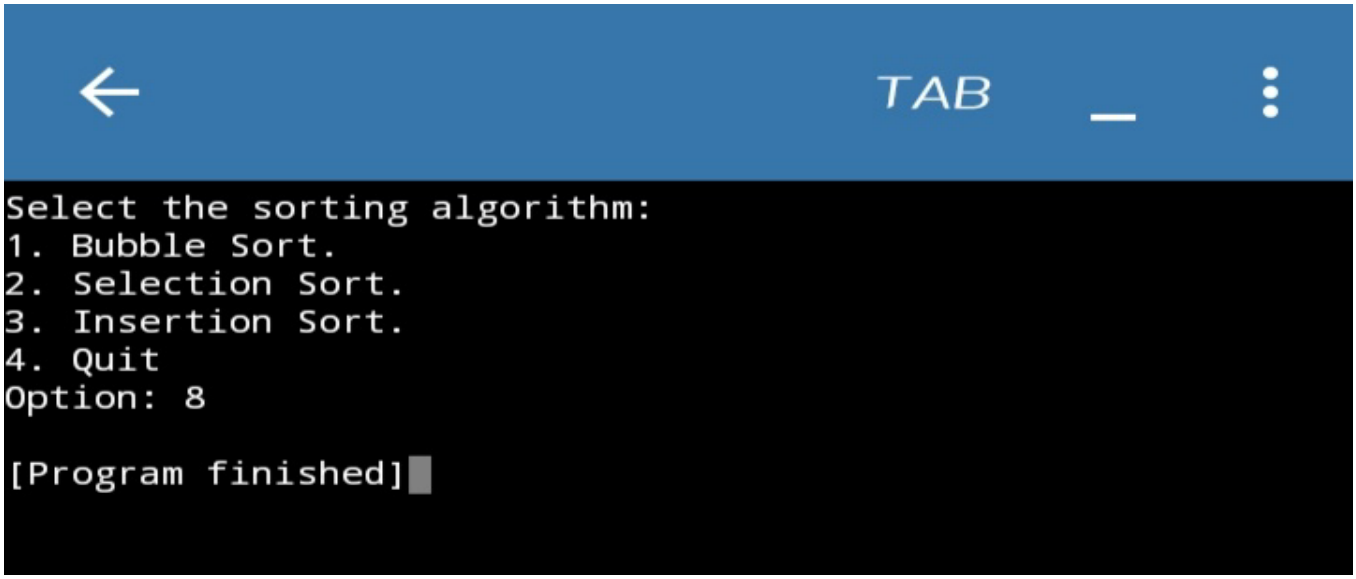
```
1  class Sorting:
2
3      def __init__(self,lst):
4          self.lst = lst
5
6      def bubble_sort(self,lst):
7          for i in range(len(lst)):
8              for j in range(len(lst)):
9                  if lst[i] < lst[j]:
10                     lst[i],lst[j] = lst[j],lst[i]
11                 else:
12                     pass
13         return lst
14
15     def selection_sort(self,lst):
16         for i in range(len(lst)):
17             smallest_element = i
18             for j in range(i+1,len(lst)):
19                 if lst[smallest_element] >
   lst[j]:
20                     smallest_element = j
21             lst[i],lst[smallest_element] =
   lst[smallest_element],lst[i]
22         return lst
23
24     def insertion_sort(self,lst):
25         for i in range(1, len(lst)):
26             index = lst[i]
27             j = i-1
28             while j >= 0 and index
```

Tab | : | ; | ' | # | (

practical_6....
/storage/emul...

```
29                  lst[j + 1] = lst[j]
30                  j -= 1
31              lst[j + 1] = index
32          return lst
33
34      def run_sort(self):
35          while True:
36              print('Select the sorting
    algorithm:')
37              print('1. Bubble Sort.')
38              print('2. Selection Sort.')
39              print('3. Insertion Sort.')
40              print('4. Quit')
41              opt = int(input('Option: '))
42              if opt == 1:
43                  print(sort.bubble_sort(self.
    lst))
44              elif opt == 2:
45                  print(sort.
    selection_sort(self.lst))
46              elif opt == 3:
47                  print(sort.
    insertion_sort(self.lst))
48              else:
49                  break
50  lst = [4,2,3,9,12,1]
51  sort = Sorting(lst)
52  sort.run_sort()
53
```

Tab | : | ; | ' | #



Practical 7
Practical 7A: Implement the following for Hashing:
Aim: a) Write a program to implement the collision technique.
Theory: Theory on Practical 7

Hashing:
Hashing is an important Data structure which is designed to use a special function called the Hash function which is used to map a given value with a particular key for faster access of elements. The efficiency of mapping depends of the efficiency of the hash function used.

Collisions:
A Hash Collision Attack is an attempt to find two input strings of a hash function that produce the same hash result. If two separate inputs produce the same hash output, it is called a collision.

Collision Techniques:
Separate Chaining:
The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

Open Addressing:
Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. So at any point, the size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed).

prac7a.py
/storage/emul...

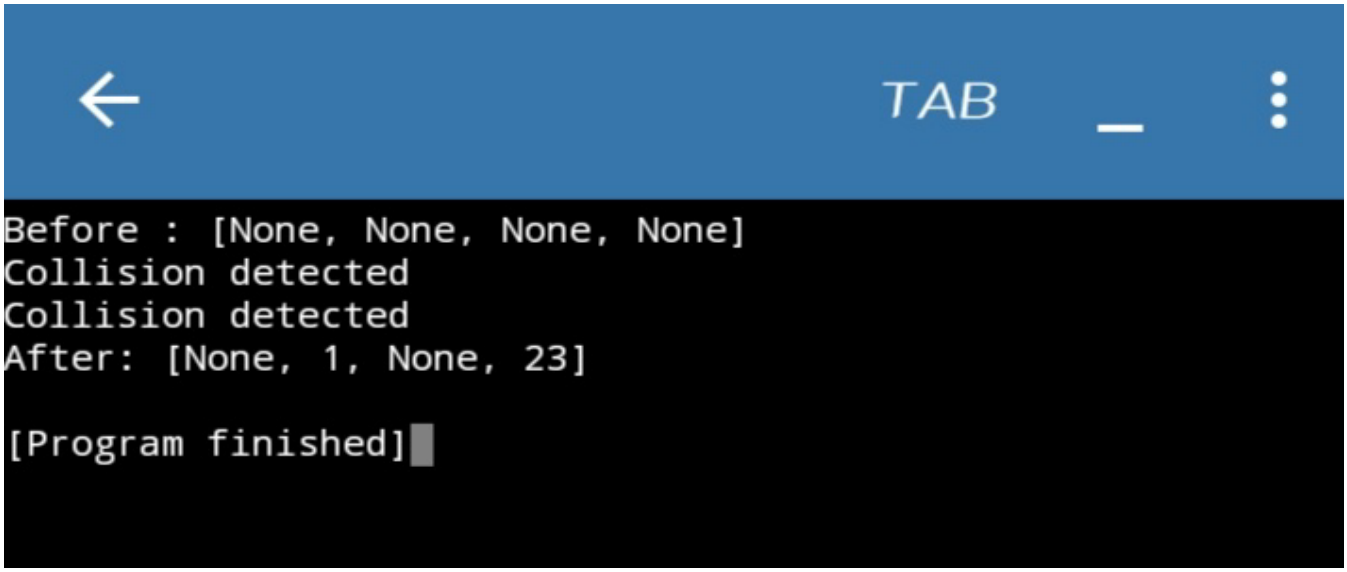
```
1  class Hash:
2      def __init__(self, keys, lowerrange,
   higherrange):
3          self.value = self.
   hashfunction(keys,lowerrange,
   higherrange)
4
5      def get_key_value(self):
6          return self.value
7
8      def hashfunction(self,keys,
   lowerrange, higherrange):
9          if lowerrange == 0 and
   higherrange > 0:
10             return keys%(higherrange)
11
12 if __name__ == '__main__':
13     list_of_keys = [23,43,1,87]
14     list_of_list_index = [None,None,
   None,None]
15     print("Before :" +
   str(list_of_list_index))
16     for value in list_of_keys:
17         #print(Hash(value,0,
   len(list_of_keys)).get_key_value())
18         list_index = Hash(value,0,
   len(list_of_keys)).get_key_value()
19         if list_of_list_index[list_in
20             print("Collision detect
21         else:
```

Tab | : | ; | ' | # | (

prac7a.py
/storage/emul...

```
9   higherrange > 0:
10          return keys%(higherrange)
11
12  if __name__ == '__main__':
13     list_of_keys = [23,43,1,87]
14     list_of_list_index = [None,None,
    None,None]
15     print("Before :" +
    str(list_of_list_index))
16     for value in list_of_keys:
17        #print(Hash(value,0,
    len(list_of_keys)).get_key_value())
18        list_index = Hash(value,0,
    len(list_of_keys)).get_key_value()
19        if list_of_list_index[list_index]:
20           print("Collision detected")
21        else:
22           list_of_list_index[list_index] =
    value
23
24     print("After:" +
    str(list_of_list_index))
25
```

Tab | : | ; | ' | # | (



Practical 7b:
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. ... Along with quadratic probing and double hashing, linear probing is a form of open addressing

prac7b.py
/storage/emul...

```
1  class Hash:
2      def __init__(self, keys: int,
   lower_range: int, higher_range: int) -
   > None:
3          self.value = self.
   hash_function(keys, lower_range,
   higher_range)
4
5      def get_key_value(self) -> int:
6          return self.value
7
8      @staticmethod
9      def hash_function(keys: int,
   lower_range: int, higher_range: int) -
   > int:
10         if lower_range == 0 and
   higher_range > 0:
11             return keys % higher_range
12
13
14 if __name__ == '__main__':
15     linear_probing = True
16     list_of_keys = [23, 43, 1, 87]
17     list_of_list_index = [None]*4
18     print("Before :" +
   str(list_of_list_index))
19     for value in list_of_keys:
20         list_index = Hash(value, 0
   len(list_of_keys)).get_key_v
21         print("Hash value for " +
```

Tab | : | ; | ' | # | (

prac7b.py
/storage/emul...

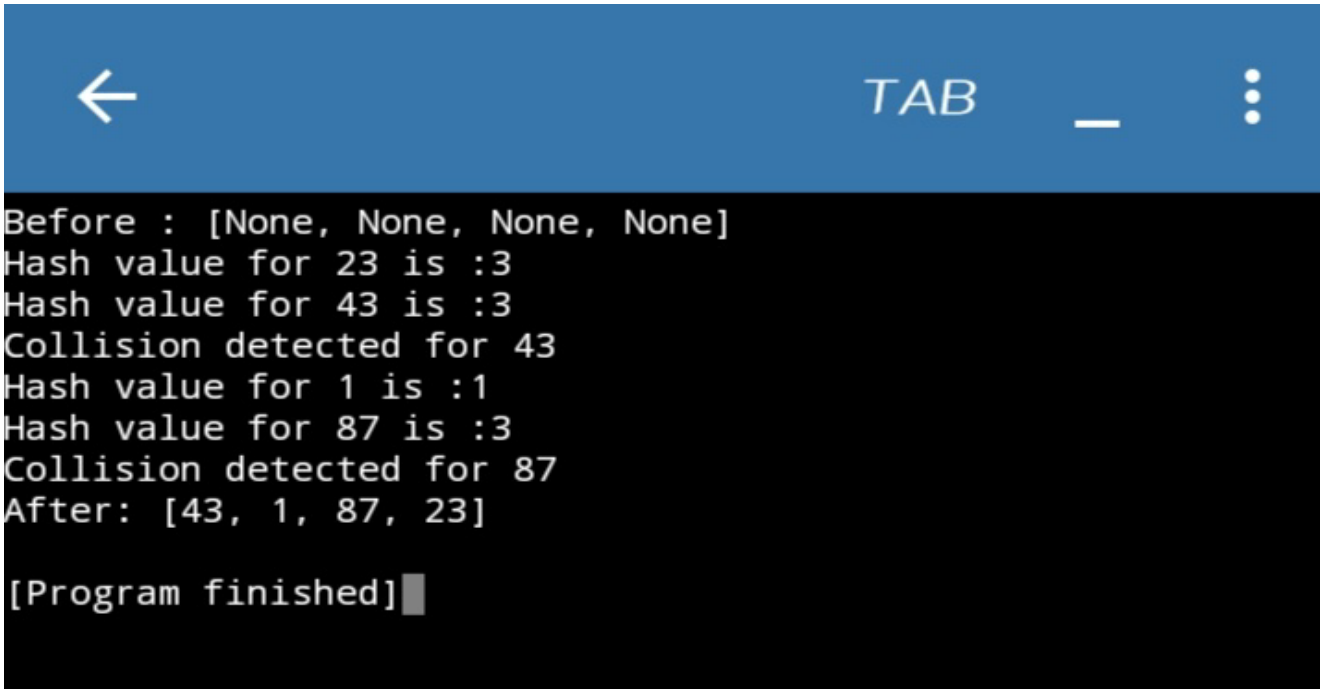
```
21  str(value) + " is :" + str(list_index))
22      if list_of_list_index[list_index]:
23          print("Collision detected for " +
    str(value))
24          if linear_probing:
25              old_list_index = list_index
26              if list_index ==
    len(list_of_list_index) - 1:
27                  list_index = 0
28              else:
29                  list_index += 1
30              list_full = False
31              while
    list_of_list_index[list_index]:
32                  if list_index ==
    old_list_index:
33                      list_full = True
34                      break
35                  if list_index + 1 ==
    len(list_of_list_index):
36                      list_index = 0
37                  else:
38                      list_index += 1
39              if list_full:
40                  print("List was full . Could
    not save")
41              else:
42
    list_of_list_index[list_index
43      else:
```

Tab | : | ; | ' | # |

prac7b.py
/storage/emul...

```
31              while
list_of_list_index[list_index]:
32                  if list_index ==
old_list_index:
33                      list_full = True
34                      break
35                  if list_index + 1 ==
len(list_of_list_index):
36                      list_index = 0
37                  else:
38                      list_index += 1
39              if list_full:
40                  print("List was full . Could
not save")
41              else:
42
list_of_list_index[list_index] = value
43          else:
44              list_of_list_index[list_index] =
value
45      print("After: " +
str(list_of_list_index))
46
```

Tab | : | ; | ' | # | (



Practical 8:
Aim: Write a program for in-order, postorder and preorder traversal of tree.
Theory:
In order:
In case of binary search trees (BST), in order traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of in order traversal where in order traversal s reversed can be used.

Preorder:
Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expression on of an expression tree.

Postorder:
Postorder traversal is also useful to get the postfix expression of an expression tree.

prac8.py
/storage/emul...

```
1  class Node:
2      def __init__(self, key):
3          self.left = None
4          self.right = None
5          self.value = key
6
7      def PrintTree(self):
8          if self.left:
9              self.left.PrintTree()
10         print(self.value)
11         if self.right:
12             self.right.PrintTree()
13
14     def Printpreorder(self):
15         if self.value:
16             print(self.value)
17             if self.left:
18                 self.left.Printpreorder()
19             if self.right:
20                 self.right.Printpreorder()
21
22     def Printinorder(self):
23         if self.value:
24             if self.left:
25                 self.left.Printinorder()
26             print(self.value)
27             if self.right:
28                 self.right.Printinorder()
29
30     def Printpostorder(self):
```

Tab | : | ; | ' | # | (

prac8.py
/storage/emul...

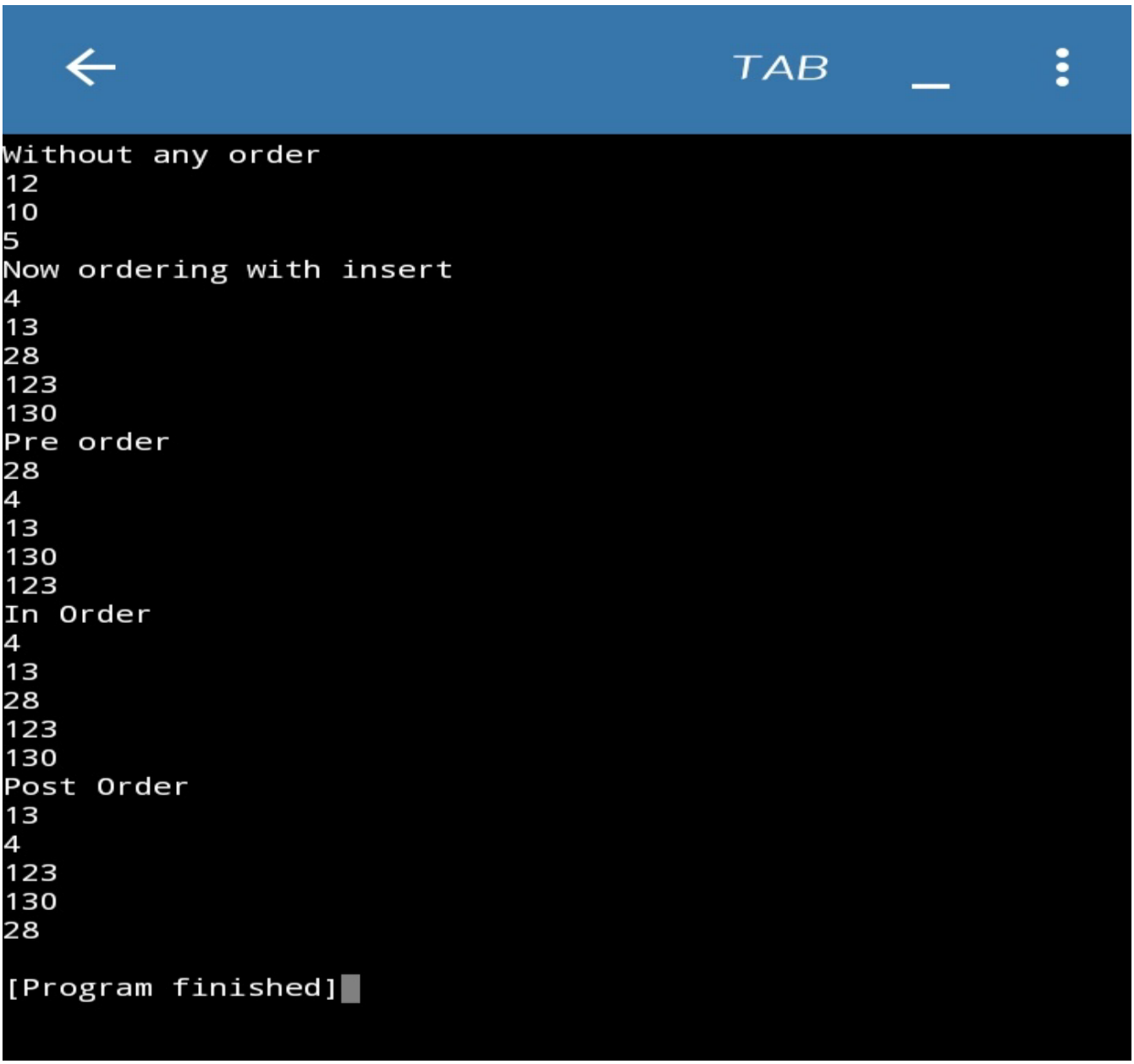
```
31          if self.value:
32              if self.left:
33                  self.left.Printpostorder()
34              if self.right:
35                  self.right.Printpostorder()
36              print(self.value)
37
38      def insert(self, data):
39          if self.value:
40              if data < self.value:
41                  if self.left is None:
42                      self.left = Node(data)
43                  else:
44                      self.left.insert(data)
45              elif data > self.value:
46                  if self.right is None:
47                      self.right = Node(data)
48                  else:
49                      self.right.insert(data)
50          else:
51              self.value = data
52
53
54  if __name__ == '__main__':
55      root = Node(10)
56      root.left = Node(12)
57      root.right = Node(5)
58      print("Without any order")
59      root.PrintTree()
60      root_1 = Node(None)
```

Tab | : | ; | ' | # | (

prac8.py
/storage/emul...

```
52
53
54  if __name__ == '__main__':
55      root = Node(10)
56      root.left = Node(12)
57      root.right = Node(5)
58      print("Without any order")
59      root.PrintTree()
60      root_1 = Node(None)
61      root_1.insert(28)
62      root_1.insert(4)
63      root_1.insert(13)
64      root_1.insert(130)
65      root_1.insert(123)
66      print("Now ordering with insert")
67      root_1.PrintTree()
68      print("Pre order")
69      root_1.Printpreorder()
70      print("In Order")
71      root_1.Printinorder()
72      print("Post Order")
73      root_1.Printpostorder()
74
```

Tab | : | ; | ' | # | (


TAB
Without any order
12
10
5
Now ordering with insert
4
13
28
123
130
Pre order
28
4
13
130
123
In Order
4
13
28
123
130
Post Order
13
4
123
130
28
[Program finished]