



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.SMIT DHURVE

Roll No: 390

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: 390****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(a)

Aim: 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.

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

Code and Output:

```
1- # Implement the following for Array:
2 # Write a program to store the elements in 1-D array and provide an
   option
3 # to perform the operations like searching, sorting, merging,
   reversing the elements.
4 arr1=[12,35,42,22,1,6,54]
5 arr2=['hello','world']
6 arr1.index(35)
7 print(arr1)
8 arr1.sort()
9 print(arr1)
10 arr1.extend(arr2)
11 print(arr1)
12 arr1.reverse()
13 print(arr1)
```

```
[12, 35, 42, 22, 1, 6, 54]
[1, 6, 12, 22, 35, 42, 54]
[1, 6, 12, 22, 35, 42, 54, 'hello', 'world']
['world', 'hello', 54, 42, 35, 22, 12, 6, 1]
>
```

Practical 1(b)

Aim: Implement the following for Array:

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.

Code and Output:

```
1  Mat1 = [[3, 4, -6],
2         [12,71,24],
3         [21,3,21]]
4
5  Mat2 = [[2, 16, -16],
6         [1,7, -3],
7         [-1,3,3]]
8  Mat3 = [[0,0,0,],
9         [0,0,0,],
10        [0,0,0,]]
11
12 # Matrix Addition
13 for i in range(len(Mat1)):
14     for j in range(len(Mat2[0])):
15         for k in range(len(Mat2)):
16             Mat3[i][j] += Mat1[i][k] + Mat2[k][j]
17
18 print(Mat3)
19
20 # Matrix Multiplication
21
22 Mat3 = [[0, 0, 0, 0],
23         [0, 0, 0, 0],
24         [0, 0, 0, 0]]
25
26 for i in range(len(Mat1)):
27     for j in range(len(Mat2[0])):
28         for k in range(len(Mat2)):
29             Mat3[i][j] += Mat1[i][k] * Mat2[k][j]
30
31 print(Mat3)
32
33 #matrix transpose
34 for i in map(list, zip(*Mat1)):
35     print(i)
36
```

```
[[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]
[Finished in 0.0s]
```

Practical 2

Aim: Implement Linked List. 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 Linked List recursively we need to divide the Linked List into two parts: head and remaining. Head points to the first element initially. Remaining points to the next element from the head. We traverse the Linked List 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

Code and Output:

```
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:
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
```

```

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.element)
56             last = last.previous
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 = self.get_tail()
85         self.get_tail().next = new_value
86         self.size += 1
87

```

```

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
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
121     def get_node_at(self, index):
122         element_node = self.head
123         counter = 0
124         if index == 0:
125             return element_node.element
126         if index > self.size - 1:
127             print("Index out of bound")
128             return None
129         while (counter < index):
130             element_node = element_node.next
131             counter += 1
132         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).previous
138
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)
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 type(value.element) == type(my_list.head):
175                 print("Searching at " + str(index) + " and value is " + str(value.element.element))
176             else:

```

```

177         print("Searching at " + str(index) + " and value is " + str(value.element))
178         if value.element == search_value:
179             print("Found value at " + str(index) + " location")
180             return True
181         index += 1
182     print("Not Found")
183     return False
184
185     def merge(self, linkedlist_value):
186         if self.size > 0:
187             last_node = self.get_node_at(self.size-1)
188             last_node.next = linkedlist_value.head
189             linkedlist_value.head.previous = last_node
190             self.size = self.size + linkedlist_value.size
191
192         else:
193             self.head = linkedlist_value.head
194             self.size = linkedlist_value.size
195
196     l1 = Node('Element 1')
197     my_list = LinkedList()
198     my_list.add_head(l1)
199     my_list.add_tail('Element 2')
200     my_list.add_tail('Element 3')
201     my_list.add_tail('Element 4')
202     my_list.get_head().element.element
203     my_list.add_between_list(2, 'Element between')
204     my_list.remove_between_list(2)
205
206     my_list2 = LinkedList()
207     l2 = Node('Element 5')
208     my_list2.add_head(l2)
209     my_list2.add_tail('Element 6')
210     my_list2.add_tail('Element 7')
211     my_list2.add_tail('Element 8')
212     my_list.merge(my_list2)
213     my_list.get_previous_node_at(3).element
214     my_list.reverse_display()
215     my_list.search('Element 6')
216

```


Practical 3(a)

Aim: Implement the following for Stack:

- a) Perform Stack operations using Array implementation.

Theory:

Stacks is one of the earliest data structures defined in computer science. In simple 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

Code and Ouput:

```
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()
31 stack.push(1)
32 stack.push(3)
33 stack.push(5)
34 stack.pop()
35 stack.display()
36 stack.get_head()
```

```
[1, 3]
[Finished in 0.0s]
```

Practical 3(b)

Aim: 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.

Code:

Output:

```
Move disk 1 from source A to destination
Move disk 2 from source A to destination
Move disk 1 from source C to destination
Move disk 3 from source A to destination
Move disk 1 from source B to destination
Move disk 2 from source B to destination
Move disk 1 from source A to destination
Move disk 4 from source A to destination
Move disk 1 from source C to destination
Move disk 2 from source C to destination
Move disk 1 from source B to destination
Move disk 3 from source C to destination
Move disk 1 from source A to destination
Move disk 2 from source A to destination
Move disk 1 from source C to destination

Process finished.
```

Practical 3(C)

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 : $4x^7 + 12x^2 + 45$

Code and Output:

```
1- class Node:
2
3-     def __init__(self, element, next=None):
4         self.element = element
5         self.next = next
6         self.previous = None
7
8-     def display(self):
9         print(self.element)
10
11
12- class LinkedList:
13
14-     def __init__(self):
15         self.head = None
16         self.size = 0
17
18-     def add_head(self, e):
19         self.head = Node(e)
20         self.size += 1
21
```

```
Enter the order for polynomial : 10
Enter coefficient for power 10 : 100
Enter coefficient for power 9 :
```

Practical 3(d)

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.

Code and Output:

```
1- def factorial(number):
2-     if number < 0:
3-         print('Invalid entry! Cannot find factorial of a negative
         number')
4-         return -1
5-     if number == 1 or number == 0:
6-         return 1
7-     else:
8-         return number * factorial(number - 1)
9
10
11- def factorial_iteration(number):
12-     if number < 0:
13-         print('Invalid entry! Cannot find factorial of a negative
         number')
14-         return -1
15-     fact = 1
16-     while(number > 0):
17-         fact = fact * number
18-         number = number - 1
19-     return fact
```

```
Factorial using Recursion of 8 is: 40320
Factorial using Iteration of 8 is: 40320
>
```

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

Code and Output:

```

1- class CircularQueue:
2-
3-     #Constructor
4-     def __init__(self):
5-         self.queue = list()
6-         self.head = 0
7-         self.tail = 0
8-         self.maxSize = 8
9-
10-    #Adding elements to the queue
11-    def enqueue(self,data):
12-        if self.size() == self.maxSize-1:
13-            return ("Queue Full!")
14-        self.queue.append(data)
15-        self.tail = (self.tail + 1) % self.maxSize
16-        return True
17-
18-    #Removing elements from the queue
19-    def dequeue(self):
20-        if self.size()==0:
21-            return ("Queue Empty!")
22-        data = self.queue[self.head]
23-        self.head = (self.head + 1) % self.maxSize

```

```

True
True
True
True
True
True
True
True
True
True
Queue Full!
Queue Full!
1
2
3
4
5
6
7
Queue Empty!
Queue Empty!
> |

```

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 searcher 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.

Code and Output:

```
1 print ("BINARY SEARCH METHOD\n")
2 def bsm(arr,start,end,num):
3     if end==start:
4         mid=start+(end-start)//2
5         if arr[mid]==x:
6             return mid
7         elif arr[mid]>x:
8             return bsm(arr,start,mid-1,x)
9         else:
10            return bsm(arr,mid+1,end,x)
11     else:
12         return -1
13 arr=[10,27,36,49,58,69,70]
14 x=int(input("Enter the number to be searched : "))
15 result=bsm(arr,0,len(arr)-1,x)
16 if result != -1:
17     print ("Number is found at ",result)
18 else:
19     print ("Number is not present\n")
20
21 print ("Linear Search\n")
22 def linearsearch(arr, x):
23     for i in range(len(arr)):
24         if arr[i] == x:
```

BINARY SEARCH METHOD

Enter the number to be searched : 27
Number is found at 1
Linear Search

enter character you want to search: 36
element found at index -1
> |

Practical 6

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

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 sub arrays 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.

Code and Output:

:

```

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 < lst[j]:
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))

```

```

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

```

```

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

```

Practical 7(a)

Aim: Implement the following for Hashing:

Write a program to implement the collision technique.

Theory:

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:** When one or more hash values compete with a single hash table slot, collisions occur. To resolve this, the next available empty slot is assigned to the current hash value
- **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)

Code:

```
1 def search_value(value,hash_table):
2     hash_value = hash_function(value,list_size)
3     print(value,hash_table[hash_value])
4     if value in hash_table[hash_value]:
5         print("Value found")
6     else:
7         print("value was not found")
8
9
10 def hash_function(value,list_size):
11     return value % list_size
12
13 def create_hash_table(main_table,hash_table):
14     for element in main_table:
15         hash_value = hash_function(element,list_size)
16         if hash_table[hash_value][0]:
17             print("collision Detected")
18             hash_table[hash_value].append(element)
19         else:
20             hash_table[hash_value][0] = element
21
22 list_size = 10
23 main_table = [45,92,13,34,75,96,71,83,69,10]
24 hash_table = [[None] for i in range(list_size)]
25 print(hash_table)
26 create_hash_table(main_table,hash_table)
27 print(hash_table)
28 search_value(71,hash_table)
29
```

Output:

```
[[None], [None], [None], [None], [None], [None], [None], [None], [None], [None]]
collision Detected
collision Detected
[[10], [71], [92], [13, 83], [34], [45, 75], [96], [None], [None], [69]]
71 [71]
Value found
```

Practical 7(b)

Aim: Implement the following for Hashing:

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.

Code:

```

1 def search_value(value, hash_table):
2     hash_value = hash_function(value, list_size)
3     if value in hash_table:
4         print("Value found")
5     else:
6         print("value was not found")
7
8
9 def hash_function(value, list_size):
10     return value % list_size
11
12
13 def create_hash_table( main_table , hash_table):
14     for element in main_table:
15         print(hash_table)
16         index = 0
17         hash_value = hash_function(element, list_size)
18         while index < list_size:
19             if hash_table[hash_value]:
20                 print("collision Detected")
21                 print("Moving to another slot -->")
22                 if hash_value == (list_size - 1):
23                     hash_value = 0
24                     index += 1
25                 else:
26                     hash_value += 1
27                     index += 1
28             else:
29                 hash_table[hash_value] = element
30                 break
31
32 list_size = 10
33 main_table = [45, 92, 13, 34, 75, 96, 71, 83, 69, 10]
34 hash_table = [None for i in range(list_size)]
35 print(hash_table)
36 create_hash_table(main_table, hash_table)
37 print(hash_table)
38 search_value(71, hash_table)
39

```

Output:

```

[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

```

Practical 8

Aim: Write a program for inorder, postorder and preorder traversal of tree.

Theory:

- Inorder: In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal is 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.

Code:

```
1 class Node:
2     def __init__(self, key):
3
4         self.left = None
5
6         self.right = None
7
8         self.val = key
9
10
11
12
13
14
15
16 # A function to do inorder tree traversal
17 def printInorder(root):
18
19
20
21
22     if root:
23
24
25
26         # First recur on left child
27         printInorder(root.left)
28
29
30
31         # then print the data of node
32         print(root.val),
33
34
35
36
37         # now recur on right child
38         printInorder(root.right)
39
40
41
42
43
```

```
14         # Then recur on left child
15
16         printPreorder(root.left)
17
18
19
20         # Finally recur on right child
21
22         printPreorder(root.right)
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999
```

Output:

```
Preorder traversal of binary tree is
1 2 4 5 3
Inorder traversal of binary tree is
4 2 5 1 3
Process finished.
```