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 DHRUVE

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 elementin an array is similar to assigning values to scalar variables. Simply reference an individual element of anarray 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:

```
# Implement the following for Array:
# 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.
arr1=[12,35,42,22,1,6,54]
arr2=['hello','world']
arr1.index(35)
print(arr1)
arr1.sort()
print(arr1)
arr1.extend(arr2)
print(arr1)
arr1.reverse()
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:

```
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)
```

```
[[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:

```
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)
                first = first.next
            print(first.element)
            first = first.next

    def reverse_display(self):
        if self.size == 0:
            print("No element")
            return None
```

```python
        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
            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
```

```python
    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

    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
```

```python

    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)
        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:
```

```python
                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)
            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')
```

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 objectsthat 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:

```
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()
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 : 4x7 + 12x2 + 45

Code and Output:

```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 add_head(self, e):
        self.head = Node(e)
        self.size += 1

```

```
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:

```
def factorial(number):
    if number < 0:
        print('Invalid entry! Cannot find factorial of a negative
            number')
        return -1
    if number == 1 or number == 0:
        return 1
    else:
        return number * factorial(number - 1)


def factorial_iteration(number):
    if number < 0:
        print('Invalid entry! Cannot find factorial of a negative
            number')
        return -1
    fact = 1
    while(number > 0):
        fact = fact * number
        number = number - 1
    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 REARto -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:

```python
class CircularQueue:

    #Constructor
    def __init__(self):
        self.queue = list()
        self.head = 0
        self.tail = 0
        self.maxSize = 8

    #Adding elements to the queue
    def enqueue(self,data):
        if self.size() == self.maxSize-1:
            return ("Queue Full!")
        self.queue.append(data)
        self.tail = (self.tail + 1) % self.maxSize
        return True

    #Removing elements from the queue
    def dequeue(self):
        if self.size()==0:
            return ("Queue Empty!")
        data = self.queue[self.head]
        self.head = (self.head + 1) % self.maxSize
```

```
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:

```
print ("BINARY SEARCH METHOD\n")
def bsm(arr,start,end,num):
    if end>=start:
        mid=start+(end-start)//2
        if arr[mid]==x:
            return mid
        elif arr[mid]>x:
            return bsm(arr,start,mid-1,x)
        else:
            return bsm(arr,mid+1,end,x)
    else:
        return -1
arr=[10,27,36,49,58,69,70]
x=int(input("Enter the number to be searched : "))
result=bsm(arr,0,len(arr)-1,x)
if result != -1:
    print ("Number is found at ",result)
else:
    print ("Number is not present\n")

print ("Linear Search\n")
def linearsearch(arr, x):
   for i in range(len(arr)):
```

```
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:

:

```
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

    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))
```

```
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
```

```
                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()
```

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:

```python
def search_value(value,hash_table):
    hash_value = hash_function(value,list_size)
    print(value,hash_table[hash_value])
    if value in hash_table[hash_value]:
        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:
        hash_value = hash_function(element,list_size)
        if hash_table[hash_value][0]:
            print("collision Detected")
            hash_table[hash_value].append(element)
        else:
            hash_table[hash_value][0] = element

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)
```

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:

```
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)
```

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 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.

Code:

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



# A function to do inorder tree traversal
def printInorder(root):

    if root:

        # First recur on left child
        printInorder(root.left)

        # then print the data of node
        print(root.val),

        # now recur on right child
        printInorder(root.right)
```

```
        # Then recur on left child
        printPreorder(root.left)


        # Finally recur on right child
        printPreorder(root.right)




# Driver code

root = Node(1)

root.left      = Node(2)

root.right     = Node(3)

root.left.left  = Node(4)

root.left.right  = Node(5)

print "Preorder traversal of binary tree is"

printPreorder(root)



print "\nInorder traversal of binary tree is"

printInorder(root)
```

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.
```