**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. BHAAVIK BHAVESH ASHAR

Roll No: 367 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:** 367

# Subject: Data Structures

INDEX

| Sr No | Date | Topic | Sign |
|---|---|---|---|
| 1 | 04/09/2020 | Implement the following for Array:<br>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.<br>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:<br>a) Perform Stack operations using Array implementation. b.<br>b) Implement Tower of Hanoi.<br>c) WAP to scan a polynomial using linked list and add two polynomials.<br>d) WAP to calculate factorial and to compute the factors of a given no.<br>(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:<br>a) Write a program to implement the collision technique.<br>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

```
class ArrayModification:

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

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

            index = lst[i]

            k = i - 1

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

            lst[k+1]  = index

        return lst

    def merge(self,lst,lst2):
         lst.extend(lst2)
         print(lst)

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

lst = [2,9,1,7,3,5,2]
lst2 = [4,6,8,9,4,5]
Arrmod = ArrayModification()
print(Arrmod.linear_search(lst,3))
print(Arrmod.merge(lst,lst2))
print(Arrmod.insertion_sort(lst))
print(Arrmod.reverse(lst))
```

# PRACTICAL 1B

```
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)
```

PRACTICAL 2

```
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(list1.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
        last = list1.get_tail()
        print(last.element)
        while last.previous:
```

```
        if type(last.previous.element) == type(list1.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

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

def get_previous_node_at(self,index):
    if index == 0:
        print('No previous value')
        return None
    return list1.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(list1.head):
                print("Searching at " + str(index) + " and value is "
+ str(value.element.element))
            else:
                print("Searching at " + str(index) + " and value is "
+ str(value.element))
            if value.element == search_value:
                print("Found value at " + str(index) + " location")
                return True
            index += 1
        print("Not Found")
        return False

    def merge(self,linkedlist_value):
        if self.size > 0:
            last_node = self.get_node_at(self.size-1)
            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')
list1 = LinkedList()
list1.add_head(l1)
list1.add_tail('element 2')
list1.add_tail('element 3')
list1.add_tail('element 4')
list1.get_head().element.element
```

```
list1.add_between_list(2,'element between')
list1.remove_between_list(2)

list2 = LinkedList()
l2 = Node('element 5')
list2.add_head(l2)
list2.add_tail('element 6')
list2.add_tail('element 7')
list2.add_tail('element 8')
list1.merge(list2)
list1.get_previous_node_at(3).element
list1.reverse_display()
list1.search('element 6')
```

# PRACTICAL 3 A

```
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(4)
stack.push(5)
stack.push(6)
stack.pop()
stack.display()
stack.get_head()
```

# PRACTICAL 3 B

```
class Stack:

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

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

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

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

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

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

    else:

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

towerOfHanoi(n, A, C, B)
```

PRACTICAL 3 C

```
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
        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

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

    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 value.element == search_value:
                return value.element
            index += 1
        print("Not Found")
        return False

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



my_list = LinkedList()
order = int(input('Enter the order for polynomial : '))
my_list.add_head(Node(int(input(f"Enter coefficient for power {order} : "))))
for i in reversed(range(order)):
    my_list.add_tail(int(input(f"Enter coefficient for power {i} : ")))

my_list2 = LinkedList()
my_list2.add_head(Node(int(input(f"Enter coefficient for power {order} :
"))))
for i in reversed(range(order)):
    my_list2.add_tail(int(input(f"Enter coefficient for power {i} : ")))

for i in range(order + 1):
    print(my_list.get_node_at(i).element + my_list2.get_node_at(i).element)
```

# PRACTICAL 3 D

```
factorial = 1
n = int(input('Enter Number: '))
for i in range(1,n+1):
    factorial = factorial * i

print(f'Factorial is : {factorial}')

fact = []
for i in range(1,n+1):
    if (n/i).is_integer():
        fact.append(i)

print(f'Factors of the given numbers is : {fact}')

factorial = 1
index = 1
n = int(input("Enter number : "))
def calculate_factorial(n,factorial,index):
    if index == n:
        print(f'Factorial is : {factorial}')
        return True
    else:
        index = index + 1
        calculate_factorial(n,factorial * index,index)
calculate_factorial(n,factorial,index)

fact = []
def calculate_factors(n,factors,index):
    if index == n+1:
        print(f'Factors of the given numbers is : {factors}')
        return True
    elif (n/index).is_integer():
        factors.append(index)
        index += 1
        calculate_factors(n,factors,index)
    else:
        index += 1
        calculate_factors(n,factors,index)

index = 1
factors = []
calculate_factors(n,factors,index)
```

# PRACTICAL 4

```
class ArrayQueue:
    """FIFO queue implementation using a Python list as underlying
storage."""
    DEFAULT_CAPACITY = 10          # moderate capacity for all new
queues

    def __init__(self):
        """Create an empty queue."""
        self._data = [None] * ArrayQueue.DEFAULT_CAPACITY
        self._size = 0
        self._front = 0
        self._back = 0

    def __len__(self):
        """Return the number of elements in the queue."""
        return self._size

    def is_empty(self):
        """Return True if the queue is empty."""
        return self._size == 0

    def first(self):
        """Return (but do not remove) the element at the front of the
queue.
        Raise Empty exception if the 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 if the queue is empty.
```

```
        """
        if self.is_empty():
            raise Empty('Queue is empty')
        back = (self._front + self._size - 1) % len(self._data)
        answer = self._data[back]
        self._data[back] = None         # help garbage collection
        self._front = self._front
        self._size -= 1
        self._back = (self._front + self._size - 1) % len(self._data)
        return answer

    def enqueueEnd(self, e):
        """Add an element to the back of queue."""
        if self._size == len(self._data):
            self._resize(2 * len(self.data))     # double the array
size
        avail = (self._front + self._size) % len(self._data)
        self._data[avail] = e
        self._size += 1
        self._back = (self._front + self._size - 1) % len(self._data)

    def enqueueStart(self, e):
        """Add an element to the start of queue."""
        if self._size == len(self._data):
            self._resize(2 * len(self._data))     # double the array
size
        self._front = (self._front - 1) % len(self._data)
        avail = (self._front + self._size) % len(self._data)
        self._data[self._front] = e
        self._size += 1
        self._back = (self._front + self._size - 1) % len(self._data)

    def _resize(self, cap):                  # we assume cap >=
len(self)
        """Resize to a new list of capacity >= len(self)."""
        old = self._data                       # keep track of
existing list
        self._data = [None] * cap              # allocate list with
new capacity
        walk = self._front
        for k in range(self._size):            # only consider
existing elements
            self._data[k] = old[walk]            # intentionally shift
indices
            walk = (1 + walk) % len(old)         # use old size as
modulus
        self._front = 0                          # front has been
realigned
        self._back = (self._front + self._size - 1) % len(self._data)

queue = ArrayQueue()
queue.enqueueEnd(1)
```

```
print(f"First Element: {queue._data[queue._front]}, Last Element:
{queue._data[queue._back]}")
queue._data
queue.enqueueEnd(2)
print(f"First Element: {queue._data[queue._front]}, Last Element:
{queue._data[queue._back]}")
queue._data
queue.dequeueStart()
print(f"First Element: {queue._data[queue._front]}, Last Element:
{queue._data[queue._back]}")
queue.enqueueEnd(3)
print(f"First Element: {queue._data[queue._front]}, Last Element:
{queue._data[queue._back]}")
queue.enqueueEnd(4)
print(f"First Element: {queue._data[queue._front]}, Last Element:
{queue._data[queue._back]}")
queue.dequeueStart()
print(f"First Element: {queue._data[queue._front]}, Last Element:
{queue._data[queue._back]}")
queue.enqueueStart(5)
print(f"First Element: {queue._data[queue._front]}, Last Element:
{queue._data[queue._back]}")
queue.dequeueEnd()
print(f"First Element: {queue._data[queue._front]}, Last Element:
{queue._data[queue._back]}")
queue.enqueueEnd(6)
print(f"First Element: {queue._data[queue._front]}, Last Element:
{queue._data[queue._back]}")
```

# PRACTICAL 5

```
def linear_search(lst,n):
        for i in range(len(lst)):
            if lst[i] == n:
                return print('Position:',i)
        return print("Number not found")

def binary_search(lst,n,start,end):
        if start <= end:
            mid = (end + start) // 2
            if lst[mid] == n:
                return print('Position:',mid)
            elif lst[mid] > n:
                return binary_search(lst,n,start,mid-1)
            else:
                return binary_search(lst,n,mid + 1,end)
        else:
            return print("Number not found")

def run():
     while True:
          print("Press 1 for linear search")
          print("Press 2 for binary search")
          print("Press 3 to exit")
          c = int(input())
          if c == 1:
               n = int(input("Enter number to search:"))
               linear_search(lst,n)
               break
          elif c == 2:
               s_lst = sorted(lst)
               n = int(input("Enter number to search:"))
               binary_search(s_lst,n,0,len(s_lst)-1)
               break
          else:
               break


lst = [26,74,12,3,48,2,37,15]
run()
```

# PRACTICAL 6

```
def bubble_sort(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]
     return lst

def insertion_sort(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 selection_sort(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 run():
     while True:
          print("Press 1 for bubble sort")
          print("Press 2 for insertion sort")
          print("Press 3 for selection sort")
          print("Press 4 to exit")
          print("List:",lst)
          c = int(input())
          if c == 1:
               print("Sorted list",bubble_sort(lst))
          elif c == 2:
               print("Sorted list",insertion_sort(lst))
          elif c == 3:
               print("Sorted list",selection_sort(lst))
          else:
               break

lst = [26,74,12,3,48,2,37,15]
run()
```

# PRACTICAL 7 A

```
class Hash:
    def __init__(self, keys: int, lower_range: int, higher_range: int) ->
None:
        self.value = self.hash_function(keys, lower_range, higher_range)

    def get_key_value(self) -> int:
        return self.value

    @staticmethod
    def hash_function(keys: int, lower_range: int, higher_range: int) -> int:
        if lower_range == 0 and higher_range > 0:
            return keys % higher_range


if __name__ == '__main__':
    linear_probing = True
    list_of_keys = [23, 43, 1, 87]
    list_of_list_index = [None]*4
    print("Before : " + str(list_of_list_index))
    for value in list_of_keys:
        list_index = Hash(value, 0, len(list_of_keys)).get_key_value()
        print("Hash value for " + str(value) + " is :" + str(list_index))
        if list_of_list_index[list_index]:
            print("Collision detected for " + str(value))
            if linear_probing:
                old_list_index = list_index
                if list_index == len(list_of_list_index) - 1:
                    list_index = 0
                else:
                    list_index += 1
                list_full = False
                while list_of_list_index[list_index]:
                    if list_index == old_list_index:
                        list_full = True
                        break
                    if list_index + 1 == len(list_of_list_index):
                        list_index = 0
                    else:
                        list_index += 1
                if list_full:
                    print("List was full . Could not save")
                else:
                    list_of_list_index[list_index] = value
        else:
            list_of_list_index[list_index] = value
    print("After: " + str(list_of_list_index))
```

# PRACTICAL 7 B

```
size_list = 6

def hash_function(val):
    global size_list
    return val%size_list

def map_hash_function(hash_return_values):
    return hash_return_values

def create_hash_table(list_values,main_list):
    for  values in list_values:
        hash_return_values = hash_function(values)
        list_index = map_hash_function(hash_return_values)
        if main_list[list_index]:
            print("collision detected")
            linear_probing(list_index,values,main_list)
        else:
            main_list[list_index]=values

def linear_probing(list_index,value,main_list):
    global size_list
    list_full = False
    old_list_index=list_index
    if list_index == size_list - 1:
        list_index = 0
    else:
        list_index += 1

    while main_list[list_index]:
        if list_index+1 == size_list:
            list_index = 0
        else:
            list_index += 1
        if list_index == old_list_index:
            list_full = True
            break
    if list_full == True:
         print("list was full. could not saved")




def search_list(key,main_list):
    #for i in range(size_list):
    val = hash_function(key)
    if main_list[val] == key:
        print("list found",val)
```

```
    else:
        print("not found")


list_values = [1,3,8,6,5,14]

main_list = [None for x in range(size_list)]
print(main_list)
create_hash_table(list_values,main_list)
print(main_list)
search_list(5,main_list)
```

# PRACTICAL 8

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

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

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

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

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

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

```
            self.value = data


if __name__ == '__main__':
    root = Node(10)
    root.left = Node(12)
    root.right = Node(5)
    print("Without any order")
    root.PrintTree()
    root_1 = Node(None)
    root_1.insert(28)
    root_1.insert(4)
    root_1.insert(13)
    root_1.insert(130)
    root_1.insert(123)
    print("Now ordering with insert")
    root_1.PrintTree()
    print("Pre order")
    root_1.Printpreorder()
    print("In Order")
    root_1.Printinorder()
    print("Post Order")
    root_1.Printpostorder()
```