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.Nishith Ramesh Poojary

Roll No: 345 Programme: BSc CS Semester: III

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

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

External Examiner

Mr. Gangashankar Singh
(Subject-In-Charge)

Date of Examination: (College Stamp)

Class: S.Y. B.Sc. CS Sem- III **Roll No: 345**

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

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.

GitHub link:

A) https://github.com/npoojary-techie/DS/blob/master/Practical%201a.py

B) https://github.com/npoojary-techie/DS/blob/master/1b.py

Theory:

Searching: Searching is a very basic necessity when you store data in different data structures. The simplest approach is to go across every element in the data structure and match it with the value you are searching for. This is known as linear search. It is inefficient and rarely used, but creating a program for it gives an idea about how we can implement some advanced search algorithms.

Sorting: Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Below we see five such implementations of sorting in python.

Merging: Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most respected algorithms. Merge sort first divides the array into equal halves and then combines them in a sorted manner.

Reversing: reverse () is an inbuilt method in Python programming language that reverses objects of list in place. Returns: The reverse () method does not return any value but reverse the given object from the list

Code (A):

```
list1 = [12,3,4,55,65,66,45,22,98]
size = len(list1)
def searching(n):
        print("SEARCHING")
        print(list1)
        if n not in list1:
                print(n,"not in the list")
        else:
                for i in range(size):
                        if list1[i]==n:
                                print("index of ", n," is ",i)

def sorting():
        print("SORTING")
        print(list1)
        for i in range(len(list1)):
                for j in range(i):
                        if list1[i] < list1[j]:
                                list1[i], list1[j] = list1[j], list1[i]
        print("sorted = ",list1)

def merging():
        print("MERGING")
        list2 = [6,3,45,23,56,67,77,65]
        print("list1 = ",list1)
        print("list2 = ",list2)
        merge = list1+list2
        print("merged list = ",merge)

def revrse():
        print("REVERSED")
        print(list1)
        size = len(list1)-1
        i = 0
        while size >0:
                list1[i], list1[size] = list1[size], list1[i]
                size -= 1
                i += 1
                if size<=i:
                        break
        print(list1)

searching(57)
sorting()
merging()
revrse()
```

Output(A):

```
File  Edit  Shell  Debug  Options  Window  Help
Python 3.8.5 (tags/v3.8.5:580fbb0, Jul 20 2020, 15:43:08) [MSC v.1926 32 bit (In
tel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
========== RESTART: C:\Users\Nishith\Desktop\Ds-practical-master\1a.py =========
SEARCHING
[12, 3, 4, 55, 65, 66, 45, 22, 98]
57 not in the list
SORTING
[12, 3, 4, 55, 65, 66, 45, 22, 98]
sorted =  [3, 4, 12, 22, 45, 55, 65, 66, 98]
MERGING
list1 =  [3, 4, 12, 22, 45, 55, 65, 66, 98]
list2 =  [6, 3, 45, 23, 56, 67, 77, 65]
merged list =  [3, 4, 12, 22, 45, 55, 65, 66, 98, 6, 3, 45, 23, 56, 67, 77, 65]
REVERSED
[3, 4, 12, 22, 45, 55, 65, 66, 98]
[98, 66, 65, 55, 45, 22, 12, 4, 3]
>>>
```

Code (B):

```
mat1 = [[1, 2], [3, 4]]
mat2 = [[1, 2], [3, 4]]
mat3 = [[0, 0], [0, 0]]

for i in range(0, 2):
   for j in range(0, 2):
       mat3[i][j] = mat1[i][j] + mat2[i][j]
       print("Addition of two matrices")
for i in range(0, 2):
  for j in range(0, 2):
       print(mat3[i][j], end = "")
       print()

mat1 = [[9, 2], [5, 3]]
mat2 = [[8, 1], [4, 2]]
mat3 = [[0, 0], [0, 0]]

for i in range(0, 2):
   for j in range(0, 2):
       mat3[i][j] = mat1[i][j] - mat2[i][j]
       print("Subtraction of two matrices")
for i in range(0, 2):
  for j in range(0, 2):
       print(mat3[i][j], end = "")
       print()
mat1 = [[9, 2], [5, 3]]
mat2 = [[8, 1], [4, 2]]
mat3 = [[0, 0], [0, 0]]

for i in range(0, 2):

   for j in range(0, 2):

       mat3[i][j] = mat1[i][j] * mat2[i][j]
       print("Multiplication of two matrices")
for i in range(0, 2):
  for j in range(0, 2):
       print(mat3[i][j], end = "")
       print()
```

Output (B):

```
========== RESTART: C:/Users/Nishith/Desktop/Ds-practical-master/1b.py
Addition of two matrices
Addition of two matrices
Addition of two matrices
Addition of two matrices
2
4
6
8
Subtraction of two matrices
Subtraction of two matrices
Subtraction of two matrices
Subtraction of two matrices
1
1
1
1
Multiplication of two matrices
Multiplication of two matrices
Multiplication of two matrices
Multiplication of two matrices
72
2
20
6
>>>
```

Practical-2

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

GitHub link: https://github.com/npoojary-techie/DS/blob/master/practical%202.py

Theory:

Linked List:

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.

Searching: Searching is a very basic necessity when you store data in different data structures. The simplest approach is to go across every element in the data structure and match it with the value you are searching for. This is known as linear search. It is inefficient and rarely used, but creating a program for it gives an idea about how we can implement some advanced search algorithms.

Sorting: Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Below we see five such implementations of sorting in python.

Merging: Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most respected algorithms. Merge sort first divides the array into equal halves and then combines them in a sorted manner.

Reversing: reverse () is an inbuilt method in Python programming language that reverses objects of list in place. Returns: The reverse () method does not return any value but reverse the given object from the list

Iteration: In Python, the iterative statements are also known as looping statements or repetitive statements. The iterative statements are used to execute a part of the program repeatedly as long as a given condition is true.

Recursion: Recursion can also be seen as self-referential function composition. We apply a function to an argument, then pass that result on as an argument to a second application of the same function, and so on.

Code:

```
class Stack():
    def __init__(self):
        self.items = ['4','0','7','6','nishith','abc']

    def end(self, item):
        self.items.append(item)
        print(item)

    def peek(self):
        if self.items:
            return self.items[-1]
        else:
            return None

    def size(self):
        if self.items:
            return len(self.items)
        else:
            return None

    def display(self):
        for i in self.items:
            print(i)

    def start(self, i):
        self.items.insert(0, i)

    def search(self, a):
        l = self.items
        for i in l:
            if i == a:
                print("found Value : ", a)
                break
        else:
            print("not found")

    def traverse(self):
        a = []
        l = self.items
        for i in l:
            a.append(i)
        print(a)
    def shoting_element(self):
        #bubble shotting
        nums=self.items
```

```
        def sort(nums):
            for i in range(len(nums) - 1, 0, -1):
                for j in range(i):
                    if nums[j] > nums[j + 1]:
                        temp = nums[j]
                        nums[j] = nums[j + 1]
                        nums[j + 1] = temp

        sort(nums)
        print(nums)
    #reverse
    def reverse(self):
        l=self.items
        print(l[::-1])

    def remove_value_from_particular_index(self,a):
        l=self.items
        l.pop(a)
        print(l)

class merge1(Stack):
    #inheritance
    def __init__(self):
        Stack.__init__(self)
        self.items1 = ['4','3','2','1','6']

    def merge(self):
        l = self.items
        l1=self.items1
        a=(l+l1)
        a.sort()
        print(a)




s = Stack()
# Inserting the values
s.end('-1')
s.start('-2')
s.start('5')
s.end('6')
s.end('7')
```

```
s.end('6')
s.end('7')
s.start('-1')
s.start('-2')
print("search the specific value : ")
s.search('-2')

print("Display the values one by one :")
s.display()
print("peek (End  Value) :", s.peek())
print("treverse the values : ")
s.traverse()
#Shotting element
print("Shotting the values : ")
s.shoting_element()
#reversing the list
print("Reversing  the values : ")
s.reverse()

print("remove value from particular index which is defined earlier")
s.remove_value_from_particular_index(0)

s1=merge1()
print("merge")
s1.merge()
```

Output:

```
---------- RESTART: C:\Users\Nishith\Desktop\Ds-practical-master\2.py ----------
-1
6
7
search the specific value :
found Value :  -2
Display the values one by one :
-2
-1
5
-2
4
0
7
6
nishith
abc
-1
6
7
peek (End  Value) : 7
treverse the values :
['-2', '-1', '5', '-2', '4', '0', '7', '6', 'nishith', 'abc', '-1', '6', '7']
Shotting the values :
['-1', '-1', '-2', '-2', '0', '4', '5', '6', '6', '7', '7', 'abc', 'nishith']
Reversing  the values :
['nishith', 'abc', '7', '7', '6', '6', '5', '4', '0', '-2', '-2', '-1', '-1']
remove value from particular index which is defined earlier
['-1', '-2', '-2', '0', '4', '5', '6', '6', '7', '7', 'abc', 'nishith']
merge
['0', '1', '2', '3', '4', '4', '6', '6', '7', 'abc', 'nishith']
>>>
```

Practical-3

Aim: Implement the following for
Stack:

a. Perform Stack operations using Array implementation.

b. Implement Tower of Hanoi (optional)

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

d. WAP to calculate factorial and

GitHub link:

a) : https://github.com/npoojary-techie/DS/blob/master/3a.py

c): https://github.com/npoojary-techie/DS/blob/master/3c.py

d): https://github.com/npoojary-techie/DS/blob/master/3d.py

Theory:

Stack Operations: In a stack the element inserted last in sequence will come out first as we can remove only from the top of the stack. Such feature is known as Last in First Out(LIFO) feature. The operations of adding and removing the elements is known as PUSH and POP. In the following program we implement it as add and and remove functions. We dclare an empty list and use the append() and pop() methods to add and remove the data elements.

Linked List:

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.

Code (a):

```
stack = []
stack.append('d')
stack.append('a')
stack.append('m')
stack.append('n')


print('Initial stack')
print(stack)


print('Elements poped from stack:')
print(stack.pop())
print(stack.pop())
print(stack.pop())

print('Stack after elements are popped:')
print(stack)
```

Output (a):

```
>>>
========== RESTART: C:\Users\Nishith\Desktop\Ds-practical-master\3a.py =
Initial stack
['d', 'a', 'm', 'n']
Elements poped from stack:
n
m
a
Stack after elements are popped:
['d']
>>>
```

Code (c):

```
def add(A, B, m, n):

        size = max(m, n);
        sum = [0 for i in range(size)]


        for i in range(0, m, 1):
                sum[i] = A[i]

        for i in range(n):
                sum[i] += B[i]
        return sum


def printPoly(poly, n):
        for i in range(n):
                print(poly[i], end = "")
                if (i != 0):
                        print("x^", i, end = "")
                if (i != n - 1):
                        print(" + ", end = "")


if __name__ == '__main__':


        A = [5, 0, 10, 6]


        B = [1, 2, 4]
        m = len(A)
        n = len(B)

        print("First polynomial is")
        printPoly(A, m)
        print("\n", end = "")
        print("Second polynomial is")
        printPoly(B, n)
        print("\n", end = "")
        sum = add(A, B, m, n)
        size = max(m, n)

        print("sum polynomial is")
        printPoly(sum, size)
```

Output (c):

```
>>>
========== RESTART: C:\Users\Nishith\Desktop\Ds-practical-master\3c.
First polynomial is
5 + 0x^ 1 + 10x^ 2 + 6x^ 3
Second polynomial is
1 + 2x^ 1 + 4x^ 2
sum polynomial is
6 + 2x^ 1 + 14x^ 2 + 6x^ 3
>>>
```

Code (d):

```
def recur_factorial(n):
   if n == 1:
       return n
   else:
       return n*recur_factorial(n-1)

num = int(input("Enter a number: "))

if num < 0:
   print("Sorry, factorial does not exist for negative numbers")
elif num == 0:
   print("The factorial of 0 is 1")
else:
   print("The factorial of",num,"is",recur_factorial(num))

def factorial(n):

        fact = 1
        for i in range(1, n + 1):
                fact = fact * i

        return fact


if __name__ == '_main_':
    print("The Factorial of", n, "is", factorial(n))

#using iteration
def fact(number):

    fact = 1

    for number in range(5, 1,-1):

        fact = fact * number
    return fact

number = int(input("Enter a number for iteration : "))

factorial = fact(number)
print("Factorial is "+str(factorial))
```

Output (d):

```
Enter a number: 5
The factorial of 5 is 120
Enter a number for iteration : 5
Factorial is 120
>>>
```

Practical-4

Aim: Perform Queues operations using Circular Array implementation.

GitHub Link: https://github.com/npoojary-techie/DS/blob/master/4.py

Theory:

Before you go ahead and read this tutorial, I highly recommend you to read our previous tutorial on Queues as we will be building off of those concepts. Circular Queues are widely used and are often tested on job interviews. A Circular Queue can be seen as an improvement over the Linear Queue because:

1. There is no need to reset Head and Tail pointers since they reset themselves. This means that once the Head or Tail reaches the end of the Queue, it resets itself to 0.
2. The Tail and Head can point to the same location - this means the Queue is empty
3. The Head can be greater than the Tail or vice-versa. This is possible because the Head and Tail pointers are allowed to cross each other.

Code:

```
class Stack():
    def __init__(self):
        self.items = [1,2,3,4,5]



    def enque(self,item):
        self.items.append(item)
        print(item)


    def deque(self):
        b= self.items
        b.pop()
        print(b)

    def traverse(self):
        a = []
        l = self.items
        for i in l:
            a.append(i)
        print(a)

s=Stack()

print("Adding the element in the queue : ")
s.enque(6)
print("initial queue : ")
s.traverse()

print("After removing an element from the queue : ")
s.deque()
```

Output:

```
>>>
========== RESTART: C:\Users\Nishith\Desktop\Ds-practical-master\4.py =
Adding the element in the queue :
6
initial queue :
[1, 2, 3, 4, 5, 6]
After removing an element from the queue :
[1, 2, 3, 4, 5]
>>>
```

Practical-5

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

GitHub Link: https://github.com/npoojary-techie/DS/blob/master/Practical-5.py

Theory:

Linear search: Linear search is a very basic and simple search algorithm. In Linear search, we search an element or value in a given array by traversing the array from the starting, till the desired element or value is found.

1. Traverse the array using a `for` loop.
2. In every iteration, compare the `target` value with the current value of the array.
    - If the values match, return the current index of the array.
    - If the values do not match, move on to the next array element.
3. If no match is found, return `-1`.

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

1. Compare x with the middle element.
2. If x matches with middle element, we return the mid index.
3. Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
4. Else (x is smaller) recur for the left half.

Code:

```
list1 = [2,4,15,15,25,21,36,53,63,62,72,74,88]
print("List = ",list1)
size = len(list1)
def binary_search(x):
    print("BINARY SEARCHING")
    low = 0
    high = len(list1) - 1
    mid = 0
    while low <= high:
        mid = (high + low) // 2
        if list1[mid] < x:
            low = mid + 1
        elif list1[mid] > x:
            high = mid - 1
        else:
            return mid
    return -1

def lsearching(n):
        print("LINEAR SEARCHING")
        if n not in list1:
                print(n,"not in the list")
        else:
                for i in range(size):
                        if list1[i]==n:
                                print("index of ", n," is ",i)

n = input("Enter (L) for Linear search and  (B) for Binary search \n ")
if n=="L" or n=="l":
        v = int(input("Enter a no. from the list1 "))
        lsearching(v)
elif n=="B" or n=="b":
        v = int(input("Enter a no. from the list1 "))
        print("index of ",v," is ",binary_search(v))
else:
        print("Invalid input")
```

Output:

```
========== RESTART: C:\Users\Nishith\Desktop\Ds-practical-master\5.py ======
List =  [2, 4, 15, 15, 25, 21, 36, 53, 63, 62, 72, 74, 88]
Enter (L) for Linear search and  (B) for Binary search
 l
Enter a no. from the list1 15
LINEAR SEARCHING
index of  15  is  2
index of  15  is  3
>>>
========== RESTART: C:\Users\Nishith\Desktop\Ds-practical-master\5.py ======
List =  [2, 4, 15, 15, 25, 21, 36, 53, 63, 62, 72, 74, 88]
Enter (L) for Linear search and  (B) for Binary search
 b
Enter a no. from the list1 36
BINARY SEARCHING
index of  36  is  6
>>>
```

Practical-6

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

GitHub link: https://github.com/npoojary-techie/DS/blob/master/Practical-6.py

Theory:

Bubble Sort: Bubble Sort is a simple algorithm which is used to sort a given set of n elements provided in form of an array with n number of elements. Bubble Sort compares all the element one by one and sort them based on their values.

If the given array has to be sorted in ascending order, then bubble sort will start by comparing the first element of the array with the second element, if the first element is greater than the second element, it will swap both the elements, and then move on to compare the second and the third element, and so on.

Selection Sort: Selection sort is conceptually the simplest sorting algorithm. This algorithm will first find the smallest element in the array and swap it with the element in the first position, then it will find the second smallest element and swap it with the element in the second position, and it will keep on doing this until the entire array is sorted.

Insertion sort: Insertion sort is the sorting mechanism where the sorted array is built having one item at a time. The array elements are compared with each other sequentially and then arranged simultaneously in some particular order. The analogy can be understood from the style we arrange a deck of cards. This sort works on the principle of inserting an element at a particular position, hence the name Insertion Sort.

Code:

```
list1 = [3,14,24,43,54,62,66,72,21,76,80]
print("List = ",list1)
n = len(list1)
def bubbleSort():
    print("Bubble Sorting")
    for i in range(n-1):
        for j in range(0, n-i-1):
            if list1[j] > list1[j+1] :
                list1[j], list1[j+1] = list1[j+1], list1[j]
    print(list1)

def SelectionSort():
        print("Selection Sorting")
        for i in range(n):
                for j in range(i):
                        if list1[i]<list1[j]:
                                list1[i],list1[j] = list1[j],list1[i]
        print(list1)

def InsertionSort():
    print("Insertion Sorting")
    for i in range(1, n):
        c = list1[i]
        j = i-1
        while j >=0 and c < list1[j] :
                list1[j+1] = list1[j]
                j -= 1
        list1[j+1] = c
    print(list1)

inp = input("Enter (B) for Bubble Sort, (S) for section Sort and (I) for Insertion Sort \n")
if inp=="B" or inp=="b":
        bubbleSort()
elif inp=="S" or inp=="s":
        SelectionSort()
elif inp=="I" or inp=="i":
        InsertionSort()
else:
        print("Invalid input")
```

Output:

```
>>>
========== RESTART: C:\Users\Nishith\Desktop\Ds-practical-master\6.py ==========
List =  [3, 14, 24, 43, 54, 62, 66, 72, 21, 76, 80]
Enter (B) for Bubble Sort, (S) for section Sort and (I) for Insertion Sort
b
Bubble Sorting
[3, 14, 21, 24, 43, 54, 62, 66, 72, 76, 80]
>>>
========== RESTART: C:\Users\Nishith\Desktop\Ds-practical-master\6.py ==========
List =  [3, 14, 24, 43, 54, 62, 66, 72, 21, 76, 80]
Enter (B) for Bubble Sort, (S) for section Sort and (I) for Insertion Sort
s
Selection Sorting
[3, 14, 21, 24, 43, 54, 62, 66, 72, 76, 80]
>>>
========== RESTART: C:\Users\Nishith\Desktop\Ds-practical-master\6.py ==========
List =  [3, 14, 24, 43, 54, 62, 66, 72, 21, 76, 80]
Enter (B) for Bubble Sort, (S) for section Sort and (I) for Insertion Sort
i
Insertion Sorting
[3, 14, 21, 24, 43, 54, 62, 66, 72, 76, 80]
>>>
```

Practical-7

Aim: 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.

GitHub Link:

A) https://github.com/npoojary-techie/DS/blob/master/collision.py

B) https://github.com/npoojary-techie/DS/blob/master/linear%20probing.py

Theory:

Collision technique: Collision Resolution Techniques in data structure are the techniques used for handling collision in hashing. Separate Chaining is a collision resolution technique that handles collision by creating a linked list to the bucket of hash table for which collision occurs.

Concept of linear probing:

Linear probing is s technique for resolving hash collisions of values of hash function.

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.

Linear probing can provide high performance because of its good locality of reference, but is more sensitive to the quality of its hash function than some other collision resolution schemes.

Code (A):

```
class Hash:
    def __init__(self, keys, lowerrange, higherrange):
        self.value = self.hashfunction(keys,lowerrange, higherrange)

    def get_key_value(self):
        return self.value

    def hashfunction(self,keys,lowerrange, higherrange):
        if lowerrange == 0 and higherrange > 0:
            return keys%(higherrange)

if __name__ == '__main__':
    list_of_keys = [23,43,1,87]
    list_of_list_index = [None,None,None,None]
    print("Before : " + str(list_of_list_index))
    for value in list_of_keys:
        #print(Hash(value,0,len(list_of_keys)).get_key_value())
        list_index = Hash(value,0,len(list_of_keys)).get_key_value()
        if list_of_list_index[list_index]:
            print("Collission detected")
        else:
            list_of_list_index[list_index] = value

    print("After: " + str(list_of_list_index))
```

Output (A):

```
====== RESTART: C:\Users\Nishith\Desktop\Ds-practical-master\collisi
Before : [None, None, None, None]
Collission detected
Collission detected
After: [None, 1, None, 23]
>>>
```

Code (B):

```
class Hash:
    def __init__(self, keys, lowerrange, higherrange):
        self.value = self.hashfunction(keys,lowerrange, higherrange)

    def get_key_value(self):
        return self.value

    def hashfunction(self,keys,lowerrange, higherrange):
        if lowerrange == 0 and higherrange > 0:
            return keys%(higherrange)

if __name__ == '__main__':
    linear_probing = True
    list_of_keys = [23,43,1,87]
    list_of_list_index = [None,None,None,None]
    print("Before : " + str(list_of_list_index))
    for value in list_of_keys:
        #print(Hash(value,0,len(list_of_keys)).get_key_value())
        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("Collission 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))
```

Output(B):

```
>>>
==== RESTART: C:\Users\Nishith\Desktop\Ds-practical-master\linear probing.p
y ===
Before : [None, None, None, None]
hash value for 23 is :3
hash value for 43 is :3
Collission detected for 43
hash value for 1 is :1
hash value for 87 is :3
Collission detected for 87
After: [43, 1, 87, 23]
>>>
```

Practical-8

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

GitHub Link: https://github.com/npoojary-techie/DS/blob/master/order.py

Theory:

Inorder Traversal: For binary search trees (BST), Inorder Traversal specifies the nodes in non-descending order. In order to obtain nodes from BST in non-increasing order, a variation of inorder traversal may be used where inorder traversal is reversed.

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

Postorder Traversal: Postorder traversal is used to get the postfix expression of an expression given

Inorder

Traverse the left sub-tree, (recursively call inorder(root -> left).

Visit and print the root node.

Traverse the right sub-tree, (recursively call inorder(root -> right).

Preorder

Visit and print the root node.

Traverse the left sub-tree, (recursively call inorder(root -> left).

Traverse the right sub-tree, (recursively call inorder(root -> right).

Postorder

Traverse the left sub-tree, (recursively call inorder(root -> left).

Traverse the right sub-tree, (recursively call inorder(root -> right).

Visit and print the root node.

Code:

```
import random

random.seed(23)

class Node:
    def __init__(self, val):
        self.val = val
        self.leftChild = None
        self.rightChild = None


def insert(root, key):
    if root is None:
        return Node(key)
    else:
        if root.val == key:
            return root
        elif root.val < key:
            root.rightChild = insert(root.rightChild, key)
        else:
            root.leftChild = insert(root.leftChild, key)

    return root

def PrintInorder(root):
    if root:
        PrintInorder(root.leftChild)
        print(root.val, end=" ")
        PrintInorder(root.rightChild)

def printPreorder(root):
    if root:
        print(root.val, end=" ")
        printPreorder(root.leftChild)
        printPreorder(root.rightChild)

def printPostorder(root):
    if root:
        printPostorder(root.leftChild)
        printPostorder(root.rightChild)
        print(root.val, end=" ")

tree = Node(20)
for i in range(10):
    insert(tree, random.randint(2, 100))

if __name__ == "__main__":
    print("inorder")
    PrintInorder(tree)
    print("\n")
    print("preorder")
    printPreorder(tree)
    print("\n")
    print("postorder")
    printPostorder(tree)
```

Output:

```
====== RESTART: C:/Users/Nishith/Desktop/Ds-practical-master/order.py =
inorder
4 12 18 20 39 41 47 50 56 69 77

preorder
20 12 4 18 39 77 41 56 50 47 69

postorder
4 18 12 47 50 69 56 41 77 39 20
>>>
```