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

<u>CERTIFICATE</u>

Name: Mr.__**Harshil Shah**_____

Roll No: ___**361**____   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:____361_____**

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 No 1(A)

Aim : Write a program to store the elements in 1-D array and provide an option to perform the operations like searching, sorting, merging, reversing the elements.

Theory:

Storing Data in Arrays. Assigning values to an 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 & Output:

```
1 ▾ class ArrayModification:
2
3 ▾     def linear_search(self,lst,n):
4 ▾         for i in range(len(lst)):
5 ▾             if lst[i] == n:
6                   return f'Position :{i}'
7             return -1
8
9 ▾     def insertion_sort(self,lst):
10 ▾         for i in range(len(lst)):
11
12             index = lst[i]
13
14             k = i - 1
15
16 ▾             while k >= 0 and lst[k] > index:
17                   lst[k + 1] = lst[k]
18                   k -= 1
19
20             lst[k+1]  = index
21
22             return lst
23
24 ▾     def merge(self,lst,lst2):
25             lst.extend(lst2)
```

```
Position :4
[2, 9, 1, 7, 3, 5, 2, 4, 6, 8, 9, 4, 5]
None
[1, 2, 2, 3, 4, 4, 5, 5, 6, 7, 8, 9, 9]
[9, 9, 8, 7, 6, 5, 5, 4, 4, 3, 2, 2, 1]
>
```

Practical 1a Github Link:

https://github.com/harshil85940/DS/blob/master/practical1a.py

Practical No 1(B)

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

Algorithm to perform matrix addition, matrix multiplication

Matrix addition:

1. Input the order of the matrix.
2. Input the matrix 1 elements.
3. Input the matrix 2 elements.
4. Repeat from i = 0 to m
5. Repeat from j = 0 to n
6. mat3[i][j] = mat1[i][j] + mat2[i][j]
7. Print mat3.

Matrix multiplication:

1. Input the order of the matrix1 (m * n).
2. Input the order of matrix2 (p * q).
3. Input the matrix 1 elements.
4. Input the matrix 2 elements.
5. Repeat from i = 0 to m
6. Repeat from j = 0 to q
7. repeat from k = 0 to p
8. sum=sum+ mat1[c][k] * mat2[k][d];
9. mat3[c][d]=sum
10. Print mat3.

Matrix Transpose

The transpose of a matrix is simply a flipped version of the original matrix. We can transpose a matrix by switching its rows with its columns. We denote the transpose of matrix AA by ATAT.

Code:

```
1  Mat1 = [[3, 4, -6],
2        [12,71,24],
3        [21,3,21]]
4
5  Mat2 = [[2, 16, -16],
6          [1,7,-3],
7          [-1,3,3]]
8  Mat3 = [[0,0,0,],
9        [0,0,0,],
10        [0,0,0,]]
11
12 # Matrix Addition
13 for i in range(len(Mat1)):
14     for j in range(len(Mat2[0])):
15         for k in range(len(Mat2)):
16             Mat3[i][j] += Mat1[i][k] + Mat2[k][j]
17
18 print(Mat3)
19
20 # Matrix Multiplication
21
22 Mat3 = [[0, 0, 0, 0],
23         [0, 0, 0, 0],
24         [0, 0, 0, 0]]
25
26  for i in range(len(Mat1)):
```

```
[[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]
>
```

Link:

https://github.com/harshil85940/DS/blob/master/practical1b.py

Practical No 2

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

Linked List:

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers A linked list is a sequence of data structures, which are connected together via links.Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list is the second most-used data structure after array

Linked List Representation

Linked list can be visualized as a chain of nodes, where every node points to the next node.

- Linked List contains a link element called first.
- Each link carries a data field(s) and a link field called next.
- Each link is linked with its next link using its next link.
- Last link carries a link as null to mark the end of the list

Types of Linked List

Following are the various types of linked list.

- **Simple Linked List** − Item navigation is forward only.
- **Doubly Linked List** − Items can be navigated forward and backward.

Basic Operations

Following are the basic operations supported by a list.

- **Insertion** − Adds an element at the beginning of the list.
- **Deletion** − Deletes an element at the beginning of the list.
- **Display** − Displays the complete list.
- **Search** − Searches an element using the given key.
- **Delete** − Deletes an element using the given key.

Insertion Operation

Adding a new node in linked list is a more than one step activity. We shall learn this with diagrams here. First, create a node using the same structure and find the location where it has to be inserted.

Deletion Operation

Deletion is also a more than one step process. We shall learn with pictorial representation. First, locate the target node to be removed, by using searching algorithms.

Reverse Operation

This operation is a thorough one. We need to make the last node to be pointed by the head node and reverse the whole linked list.

Code & Output:

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

```
element 8
element 7
element 6
element 5
element 4
element 3
element 2
element 1
Searching at 0 and value is element 1
Searching at 1 and value is element 2
Searching at 2 and value is element 3
Searching at 3 and value is element 4
Searching at 4 and value is element 5
Searching at 5 and value is element 6
Found value at 5 location
>
```

Link:

https://github.com/harshil85940/DS/blob/master/prac2.py

Practical No 3(A)

a) *Aim:* Perform Stack operations using Array implementation. b.

Array implementation of Stack

In array implementation, the stack is formed by using the array. All the operations regarding the stack are performed using arrays. Lets see how each operation can be implemented on the stack using array data structure.

Adding an element onto the stack (push operation)

Adding an element into the top of the stack is referred to as push operation. Push operation involves following two steps.

1. Increment the variable Top so that it can now refere to the next memory location.
2. Add element at the position of incremented top. This is referred to as adding new element at the top of the stack.

Stack is overflown when we try to insert an element into a completely filled stack therefore, our main function must always avoid stack overflow condition.

Deletion of an element from a stack (Pop operation)

Deletion of an element from the top of the stack is called pop operation. The value of the variable top will be incremented by 1 whenever an item is deleted from the stack. The top most element of the stack is stored in an another variable and then the top is decremented by 1. the operation returns the deleted value that was stored in another variable as the result.The underflow condition occurs when we try to delete an element from an already empty stack.

Code:

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

```
[4, 5]
>
```

Practical 3a Github Link:

https://github.com/harshil85940/DS/blob/master/prac3(a).py

Practical No 3(B)

Aim: Implement Tower of Hanoi.

Theory:

Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following   simple   rules:
1)   Only   one   disk   can   be   moved   at   a   time.
2) Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on   a   stack.
3) No disk may be placed on top of a smaller disk.

Algorithm

To write an algorithm for Tower of Hanoi, first we need to learn how to solve this problem with lesser amount of disks, say → 1 or 2. We mark three towers with name, source, destination and aux (only to help moving the disks). If we have only one disk, then it can easily be moved from source to destination peg.

If we have 2 disks −

- First, we move the smaller (top) disk to aux peg.
- Then, we move the larger (bottom) disk to destination peg.
- And finally, we move the smaller disk from aux to destination peg.

So now, we are in a position to design an algorithm for Tower of Hanoi with more than two disks. We divide the stack of disks in two parts. The largest disk (n^{th} disk) is in one part and all other (n-1) disks are in the second part.

Our ultimate aim is to move disk n from source to destination and then put all other (n1) disks onto it. We can imagine to apply the same in a recursive way for all given set of disks.

Code:

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

```
Enter the number of the disk in rod A : 2
['disk 1']
['disk 2']
['disk 2', 'disk 1']
> |
```

Practical 3b Github Link:

https://github.com/harshil85940/DS/blob/master/practical_3b.py

Practical No 3(c)

Aim: Write a Program to scan a polynomial using linked list and add two polynomials

Theory:

Adding two polynomials using Linked List

Given two polynomial numbers represented by a linked list. Write a function that add these lists means add the coefficients who have same variable powers.

Example:

Input:

   1st number = $5x^2 + 4x^1 + 2x^0$

   2nd number = $5x^1 + 5x^0$

Output:

   $5x^2 + 9x^1 + 7x^0$

Input:

   1st number = $5x^3 + 4x^2 + 2x^0$

   2nd number = $5x^1 + 5x^0$

Output:

   $5x^3 + 4x^2 + 5x^1 + 7x^0$

Code:

```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 _len_(self):
        return self.size

    def get_head(self):
        return self.head


    def is_empty(self):
        return self.size == 0
```

```
Enter the order for polynomial : 1
Enter coefficient for power 1 : 2
Enter coefficient for power 0 : 3
Enter coefficient for power 1 : 4
Enter coefficient for power 0 : 5
6
8
> |
```

Practical 3c Github Link:

https://github.com/harshil85940/DS/blob/master/prac3(c).py
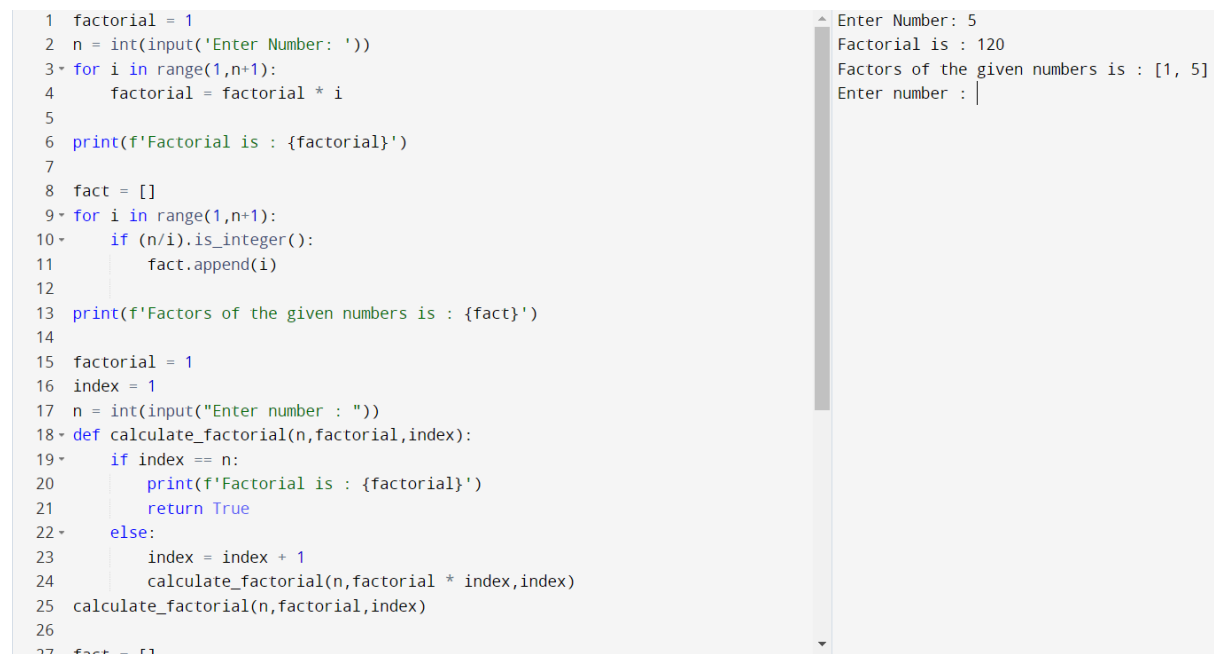
Practical No 3(d)

Aim: Write a Program to calculate factorial and to compute the factors of a given number a) using recursion , b)using iteration

Theory: Factorial of a non-negative integer, is multiplication of all integers smaller than or equal to n. For example factorial of 6 is 6*5*4*3*2*1 which is 720.Any recursive function can be written as an iterative function (and vise versa). Here is the math-like definition of recursion (again):

factorial(0) = 1

factorial(N) = N * factorial(N-1)

Code:

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

```
Enter Number: 5
Factorial is : 120
Factors of the given numbers is : [1, 5]
Enter number : |
```

Practical 3d Github Link:

https://github.com/harshil85940/DS/blob/master/prac3(d).py

Practical No 4

Aim: Perform Queues operations using Circular Array implementation.

Theory:

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

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:

```python
class ArrayQueue:

    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]
```

```
First Element: 1, Last Element: 1
First Element: 1, Last Element: 2
First Element: 2, Last Element: 2
First Element: 2, Last Element: 3
First Element: 2, Last Element: 4
First Element: 3, Last Element: 4
First Element: 5, Last Element: 4
First Element: 5, Last Element: 3
First Element: 5, Last Element: 6
>
```

Practical 4 Github Link:

https://github.com/harshil85940/DS/blob/master/prac4.py

Practical No 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:

Linear search is the simplest searching algorithm that searches for an element in a list in sequential order. We start at one end and check every element until the desired element is not found.

Linear Search Algorithm

LinearSearch(array, key)

  for each item in the array

    if item == value

      return its index

Binary Search:

Binary Search is a searching algorithm for finding an element's position in a sorted array.In this approach, the element is always searched in the middle of a portion of an array.Binary search can be implemented only on a sorted list of items. If the elements are not sorted already, we need to sort them first.

 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.

Code:

```python
class Array:

    def __init__(self,array,number):
        self.lst = sorted(array)
        self.number = number
    def binary_search(self,lst,n,start,end):

        if start <= end:
            mid = (end + start) // 2
            if lst[mid] == n:

                return f'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 -1


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

```
Select the searching algorithm:
1. Linear Search.
2. Binary Search.
3. quit.
Option: 1
Position :3
Select the searching algorithm:
1. Linear Search.
2. Binary Search.
3. quit.
Option: |
```

Practical 5 Github Link:

https://github.com/harshil85940/DS/blob/master/prac5.py

Practical No 6

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

Theory:

Insertion Sort

Insertion sort is a simple sorting algorithm that works similar to the way you sort playing cards in your hands. The array is virtually split into a sorted and an unsorted part. Values from the unsorted part are picked and placed at the correct position in the sorted part.

Algorithm

To sort an array of size n in ascending order:
1: Iterate from arr[1] to arr[n] over the array.
2: Compare the current element (key) to its predecessor.
3: If the key element is smaller than its predecessor, compare it to the elements before. Move the greater elements one position up to make space for the swapped element.

Bubble Sort

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of $O(n^2)$ where **n** is the number of items.

Selection sort

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of $O(n^2)$, where **n** is the number of items.

Code:

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

```
Select the sorting algorithm:
1. Bubble Sort.
2. Selection Sort.
3. Insertion Sort.
4. Quit
Option: 1
[2, 5, 12, 12, 43, 53, 57, 87, 98]
Select the sorting algorithm:
1. Bubble Sort.
2. Selection Sort.
3. Insertion Sort.
4. Quit
Option: |
```

Practical 6 Github Link:

https://github.com/harshil85940/DS/blob/master/prac6.py

Practical No 7(A)

Aim: Write a program to implement the collision technique

Theory:

Hashing is a data structure that is used to store a large amount of data, which can be accessed in O(1) time by operations such as search, insert and delete

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.

- It should always map large keys to small keys.
- It should always generate values between 0 to m-1 where m is the size of the hash table.
- It should uniformly distribute large keys into hash table slots.

Collision Handling

If we know the keys beforehand, then we have can have perfect hashing. In perfect hashing, we do not have any collisions. However, If we do not know the keys, then we can use the following methods to avoid collisions:

- Chaining
- Open Addressing (Linear Probing, Quadratic Probing, Double Hashing)

Chaining

While hashing, the hashing function may lead to a collision that is two or more keys are mapped to the same value. Chain hashing avoids collision. The idea is to make each cell of hash table point to a linked list of records that have same hash function value.

Code:

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

```
Before : [None, None, None, None]
Hash value for 23 is :3
Hash value for 43 is :3
Collision detected for 43
Hash value for 1 is :1
Hash value for 87 is :3
Collision detected for 87
After: [43, 1, 87, 23]
>
```

Practical 7a Github Link:

https://github.com/harshil85940/DS/blob/master/practical_7a.py

Practical No 7(B)

Aim: Write a program to implement the concept of linear probing.

Theory:

Linear probing is a scheme in computer programming for resolving hash collisions of values of hash functions by sequentially searching the hash table for a free location. This is accomplished using two values - one as a starting value and one as an interval between successive values in modular arithmetic. The second value, which is the same for all keys and known as the stepsize, is repeatedly added to the starting value until a free space is found, or the entire table is traversed.

As we can see, it may happen that the hashing technique is used to create an already used index of the array. In such a case, we can search the next empty location in the array by looking into the next cell until we find an empty cell. This technique is called linear probing.

Code:

```
1   size_list = 6
2
3 * def hash_function(val):
4       global size_list
5       return val%size_list
6
7 * def map_hash_function(hash_return_values):
8       return hash_return_values
9
10 * def create_hash_table(list_values,main_list):
11 *     for   values in list_values:
12           hash_return_values = hash_function(values)
13           list_index = map_hash_function(hash_return_values)
14 *         if main_list[list_index]:
15               print("collision detected")
16               linear_probing(list_index,values,main_list)
17 *         else:
18               main_list[list_index]=values
19
20 * def linear_probing(list_index,value,main_list):
21       global size_list
22       list_full = False
23       old_list_index=list_index
24 *     if list_index == size_list - 1:
25           list_index = 0
```

```
[None, None, None, None, None, None]
collision detected
[6, 1, 8, 3, None, 5]
list found 5
> |
```

Practical 7b Github Link:

https://github.com/harshil85940/DS/blob/master/prac7b

Practical No 8

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

Theory:

Tree Traversals

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.

Inorder Traversal

Algorithm Inorder(tree)

1. Traverse the left subtree, i.e., call Inorder(left-subtree)

2. Visit the root.

3. Traverse the right subtree, i.e., call Inorder(right-subtree)

Uses of 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. Example: Inorder traversal for the above-given figure is 4 2 5 1 3.

Preorder Traversal (Practice):

Algorithm Preorder(tree)

1. Visit the root.

2. Traverse the left subtree, i.e., call Preorder(left-subtree)

3. Traverse the right subtree, i.e., call Preorder(right-subtree)

Uses   of   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.

Example: Preorder traversal for the above given figure is 1 2 4 5 3.

Postorder Traversal

Algorithm Postorder

1. Traverse the left subtree, i.e., call Postorder(left-subtree)

2. Traverse the right subtree, i.e., call Postorder(right-subtree)

3. Visit the root.

Uses of Postorder

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

Example: Postorder traversal for the above given figure is 4 5 2 3 1.

Code:

```
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)
```

```
Without any order
12
10
5
Now ordering with insert
4
13
28
123
130
Pre order
28
4
13
130
123
In Order
4
13
28
123
130
Post Order
13
4
123
```

Practical 8 Github Link:

https://github.com/harshil85940/DS/blob/master/practical_8