



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. Aman yadav\_\_\_\_\_

Roll No: \_365\_\_\_\_\_ 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: 365\_\_\_\_\_

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'S DATA STRUCTURE

Name: Aman yadav

Oldrollno: 3106

Newrollno: 365

### 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.

#### Searching -

Searching is the process of finding a given value position in a list of values. It decides whether a search key is present in the data or not. It is the algorithmic process of finding a particular item in a collection of items. It can be done on internal data structure or on external data structure. Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored.

To search an element in a given array, it can be done in following ways:

#### Sequential search -

- Sequential search is also called as Linear Search.
- Sequential search starts at the beginning of the list and checks every element of the list.
- It is a basic and simple search algorithm.
- Sequential search compares the element with all the other elements given in the list. If the element is matched, it returns the value index, else it returns -1.

#### Binary search -

- Binary Search is used for searching an element in a sorted array.
- It is a fast search algorithm with run-time complexity of  $O(\log n)$ .
- Binary search works on the principle of divide and conquer.
- This searching technique looks for a particular element by comparing the middle most element of the collection.
- It is useful when there are large number of elements in an array.

### Sorting-

Sorting is a process of ordering or placing a list of elements from a collection in some kind of order. It is nothing but storage of data in sorted order. Sorting can be done in ascending and descending order. It arranges the data in a sequence which makes searching easier.

### Bubble sort -

- Bubble sort is a type of sorting.
- It is used for sorting 'n' (number of items) elements.
- It compares all the elements one by one and sorts them based on their values.

### Insertion sort -

- Insertion sort is a simple sorting algorithm.
- This sorting method sorts the array by shifting elements one by one.
- It builds the final sorted array one item at a time.
- Insertion sort has one of the simplest implementation.
- This sort is efficient for smaller data sets but it is insufficient for larger lists.
- It has less space complexity like bubble sort.
- It requires single additional memory space.
- Insertion sort does not change the relative order of elements with equal keys because it is stable.

### Selection sort -

- Selection sort is a simple sorting algorithm which finds the smallest element in the array and exchanges it with the element in the first position.
- Then finds the second smallest element and exchanges it with the element in the second position and continues until the entire array is sorted.

### 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.

Code.

```

class ListOperations:
    def __init__(self, lst):
        self.lst = lst

    def binary_search(self, ele):
        sorted_lst = self.bubble_sort()
        start = 0
        end = len(sorted_lst) - 1
        while start <= end:
            mid = (end + start) // 2
            if sorted_lst[mid] < ele:
                start = mid + 1
            elif sorted_lst[mid] > ele:
                end = mid - 1
            else:
                return mid
        return False

    def linear_search(self, ele):
        for i in range(len(self.lst)):
            if self.lst[i] == ele:
                return i
        return False

    def bubble_sort(self):
        sorted_lst = self.lst.copy()
        for i in range(len(sorted_lst) - 1):
            for j in range(0, len(sorted_lst) - i - 1):
                if sorted_lst[j] > sorted_lst[j + 1]:
                    sorted_lst[j], sorted_lst[j + 1] = sorted_lst[j + 1], sorted_lst[j]
        return sorted_lst

    def selection_sort(self):
        sorted_lst = self.lst.copy()
        for i in range(len(sorted_lst)):
            min_idx = i
            for j in range(i + 1, len(sorted_lst)):
                if sorted_lst[min_idx] > sorted_lst[j]:
                    min_idx = j
            sorted_lst[i], sorted_lst[min_idx] = sorted_lst[min_idx], sorted_lst[i]
        return sorted_lst

    def insertion_sort(self):
        sorted_lst = self.lst.copy()

```

```

class ListOperations:
    def __init__(self, lst):
        self.lst = lst

    def binary_search(self, ele):
        sorted_lst = self.bubble_sort()
        start = 0
        end = len(sorted_lst) - 1
        while start <= end:
            mid = (end + start) // 2
            if sorted_lst[mid] < ele:
                start = mid + 1
            elif sorted_lst[mid] > ele:
                end = mid - 1
            else:
                return mid
        return False

    def linear_search(self, ele):
        for i in range(len(self.lst)):
            if self.lst[i] == ele:
                return i
        return False

    def bubble_sort(self):
        sorted_lst = self.lst.copy()
        for i in range(len(sorted_lst) - 1):
            for j in range(0, len(sorted_lst) - i - 1):
                if sorted_lst[j] > sorted_lst[j + 1]:
                    sorted_lst[j], sorted_lst[j + 1] = sorted_lst[j + 1], sorted_lst[j]
        return sorted_lst

    def selection_sort(self):
        sorted_lst = self.lst.copy()
        for i in range(len(sorted_lst)):
            min_idx = i
            for j in range(i + 1, len(sorted_lst)):
                if sorted_lst[min_idx] > sorted_lst[j]:
                    min_idx = j
            sorted_lst[i], sorted_lst[min_idx] = sorted_lst[min_idx], sorted_lst[i]
        return sorted_lst

    def insertion_sort(self):
        sorted_lst = self.lst.copy()

```

Output--

```
Python 3.9.0 (tags/v3.9.0:9cf6752, Oct 5 2020, 15:34:40) [MSC v.1927 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: E:/Ds/Practical 1a.py =====
False
False
False
False
[11, 20, 37, 55, 77, 89, 94]
[11, 20, 37, 55, 77, 89, 94]
[11, 20, 37, 55, 77, 89, 94]
[37, 20, 55, 89, 77, 11, 94, 38, 21, 56, 90, 78, 12, 95]
[94, 11, 77, 89, 55, 20, 37]
>>> |
```

---

## Practical

no:1b

Theory.

**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.  $\text{mat3}[i][j] = \text{mat1}[i][j] + \text{mat2}[i][j]$
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.  $\text{sum} = \text{sum} + \text{mat1}[c][k] * \text{mat2}[k][d]$ ; 9.  $\text{mat3}[c][d] = \text{sum}$  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.

```
pr2.py - G:\harsh1\pr2.py (3.8.4)
File Edit Format Run Options Window Help
print("Matrix operation")
print("1.Addition")
print("2.Multiplication")
print("3.Transpose")
p = [[12,3,21],[2,43,5],[5,32,53]]
q = [[11,4,6],[21,7,9],[4,8,49]]
result = [[0,0,0],[0,0,0],[0,0,0]]
ch = int(input("Select operation :"))
if ch == 1:
    for i in range(len(p)):
        for j in range(len(q[0])):
            result[i][j] = p[i][j] + q[i][j]
        for r in result:
            print(r)
elif ch == 2:
    for i in range(len(p)):
        for j in range(len(q[0])):
            for k in range(len(q)):
                result[i][j] = p[i][k] * q[k][j]
            for r in result:
                print(r)
elif ch == 3:
    for i in range(len(p)):
        for j in range(len(q[0])):
            result[j][i] = p[i][j]
        for r in result:
            print(r)
else:
    print("Invalid selection")
```

Output.

```
===== RESTART: G:/harshl/pr2.py =====
Matrix operation
1.Addition
2.Multiplication
3.Transpose
Select operation :2
[12, 12, 126]
[42, 301, 45]
[20, 256, 2279]
>>> |
```

```
===== RESTART: G:/harshl/pr2.py =====
Matrix operation
1.Addition
2.Multiplication
3.Transpose
Select operation :3
[12, 2, 5]
[3, 43, 32]
[21, 5, 53]
>>>
```

```
===== RESTART: G:/harshl/pr2.py =====
Matrix operation
1.Addition
2.Multiplication
3.Transpose
Select operation :1
[23, 7, 27]
[23, 50, 14]
[9, 40, 96]
>>>
```

## 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 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.

Simple Linked List – Item navigation is forward only.

Doubly Linked List – Items can be navigated forward and backward.

### Operation of linked 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.

## CODE.

```
pr2.py - G:\harsh1\pr2.py (3.8.4)
File Edit Format Run Options Window Help

class Node:
    def __init__(self, element, next=None, prev=None):
        self.element = element
        self.next = next
        self.prev = prev

    def display(self):
        print(self.element)

class LinkedList:
    def __init__(self):
        self.start = None
        self.end = None
        self.length = 0

    def is_empty(self) -> bool:
        return self.length == 0

    def get_size(self) -> int:
        return self.length

    def display(self):
        if self.is_empty():
            print("Doubly Linked List is empty")
            return
        first = self.start
        print("The List: ", end='')
        print("[" + first.element, end='')
        first = first.next
        while first:
            print(", " + first.element, end='')
            first = first.next
        print("]")

    def add_head(self, e):
        old_head = self.start
        self.start = Node(e)
        self.start.next = old_head
        if old_head is not None:
            old_head.prev = self.start
        if self.end is None:
            self.end = self.start

    def get_head(self) -> Node:
        return self.start

    def remove_head(self):
        if self.is_empty():
            print("Doubly Linked List is empty")
            return
        elif self.length == 1:
            self.start = None
            self.end = None
        else:
            self.start = self.start.next
            self.length -= 1

    def add_tail(self, e):
        new_value = Node(e)
        if self.get_tail() is not None:
            old_tail = self.end
            self.end = new_value
            self.end.prev = old_tail
            old_tail.next = self.end
            self.length += 1
        else:
            self.add_head(e)

    def get_tail(self) -> Node:
        return self.end

    def remove_tail(self):
        if self.is_empty():
            print("Doubly Linked List is empty")
            return
        elif self.length == 1:
            self.start = None
            self.end = None
        else:
            self.start = self.start.next
            self.length -= 1
```

```
pr2.py - G:\harsh1\pr2.py (3.8.4)
File Edit Format Run Options Window Help

    def get_head(self) -> Node:
        return self.start

    def remove_head(self):
        if self.is_empty():
            print("Doubly Linked List is empty")
            return
        elif self.length == 1:
            self.start = None
            self.end = None
        else:
            self.start = self.start.next
            self.length -= 1

    def add_tail(self, e):
        new_value = Node(e)
        if self.get_tail() is not None:
            old_tail = self.end
            self.end = new_value
            self.end.prev = old_tail
            old_tail.next = self.end
            self.length += 1
        else:
            self.add_head(e)

    def get_tail(self) -> Node:
        return self.end

    def remove_tail(self):
        if self.is_empty():
            print("Doubly Linked List is empty")
            return
        elif self.length == 1:
            self.start = None
            self.end = None
        else:
            self.start = self.start.next
            self.length -= 1
```

## Output.

```
===== RESTART: G:/harshi/pr2.py =====
True
The List: [zero, one, two]
False
The List: [zero, two]
The List: [zero, one, two]
Head: first_name
Tail: last_name
first_name
'NoneType' object has no attribute 'display'
No head found
Doubly Linked List is empty
The List: [zero, one, two, three]
three
three
three
The List: [zero, one, two, three]
Head: zero
Tail: three
The List: [0, 1, 2, 4]
Head: 0
Tail: 4
The List: [zero, one, two, three, 0, 1, 2, 4]
Head: zero
Tail: 4
Size: 8
The List: [zero, one, two, three, 0, 1, 2, 4]
The List: [4, 2, 1, 0, three, two, one, zero]
>>>
```

## Practical no

:3a

**Aim:** Perform Stack operations using Array implementation.

Array implementation of stack :

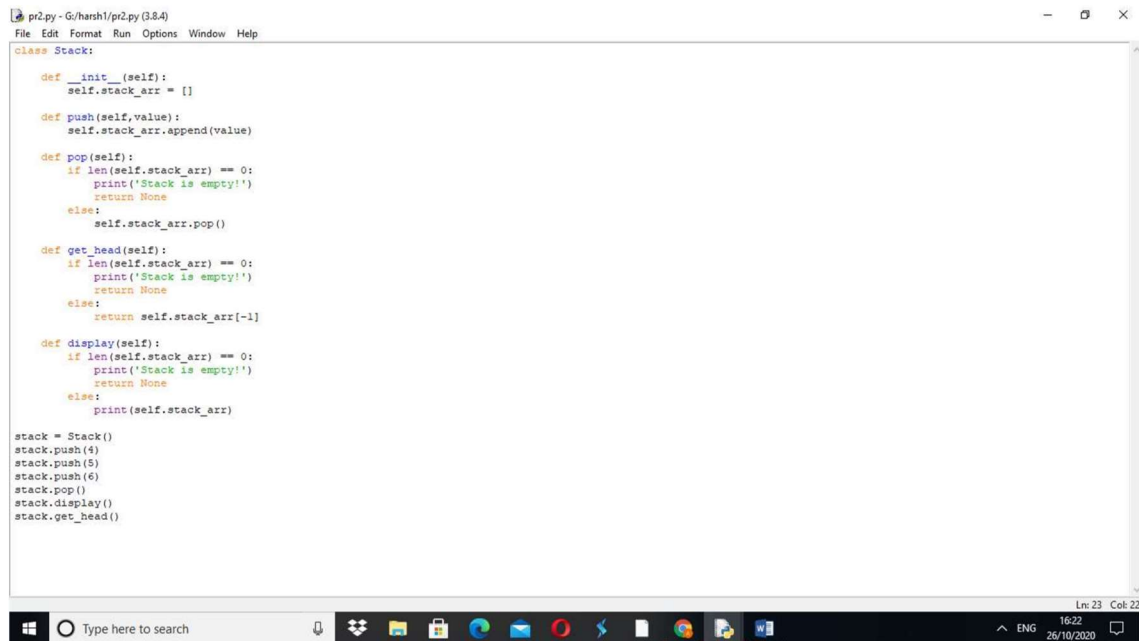
In array implementation, the stack is formed by using the array. All the operations regarding the stack are performed using arrays. Lets see how each operation can be implemented on the stack using array data structure.

Adding an element 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 refer 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.

Code.



```
pr2.py - G:/harsh1/pr2.py (3.8.4)
File Edit Format Run Options Window Help

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()

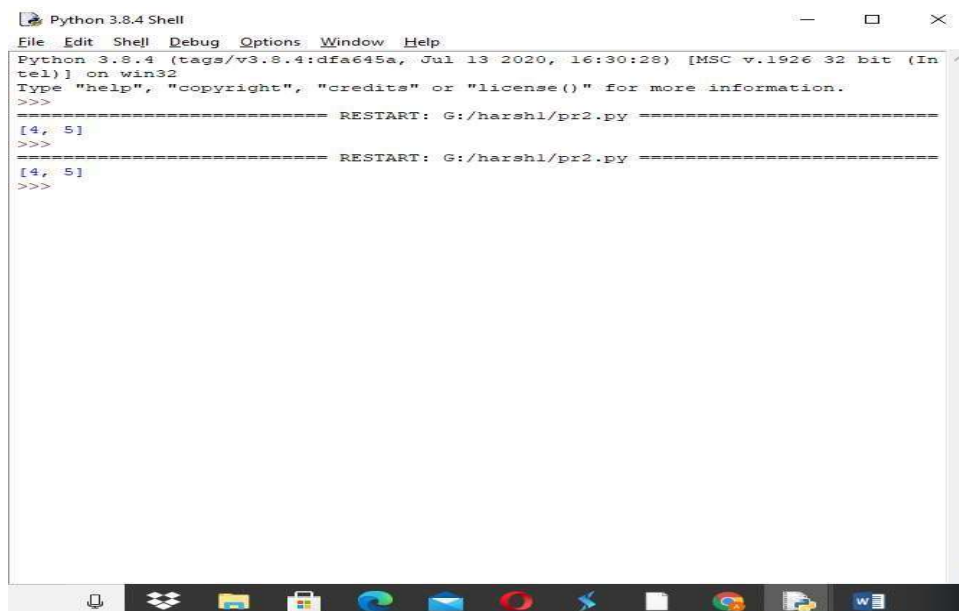
    def get_head(self):
        if len(self.stack_arr) == 0:
            print('Stack is empty!')
            return None
        else:
            return self.stack_arr[-1]

    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()

Ln: 23 Col: 22
```

Output.



```
Python 3.8.4 Shell
File Edit Shell Debug Options Window Help

Python 3.8.4 (tags/v3.8.4:dfa645a, Jul 13 2020, 16:30:28) [MSC v.1926 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: G:/harsh1/pr2.py =====
[4, 5]
>>>
===== RESTART: G:/harsh1/pr2.py =====
[4, 5]
>>>
```

### Practical no:3(b)

**Aim:** Implement Tower of Hanoi.

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.

## Algorithm:

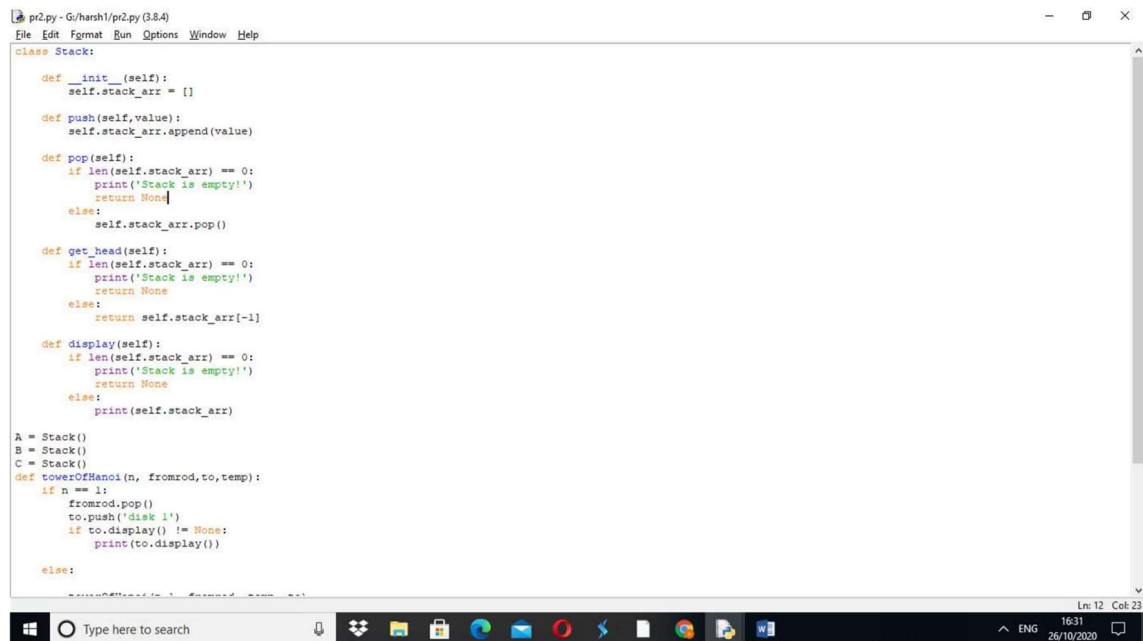
To write an algorithm for Tower of Hanoi, first we need to learn how to solve this problem with lesser amount of disks, say  $\rightarrow$  1 or 2. We mark three towers with name, source, destination and aux (only to help moving the disks). If we have only one disk, then it can easily be moved from source 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 ( $n^{\text{th}}$  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 ( $n-1$ ) disks onto it. We can imagine to apply the same in a recursive way for all given set of disks.

## Code.



```
pr2.py - G:\harsh1\pr2.py (3.8.4)
File Edit Format Run Options Window Help

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('disk 1')
        if to.display() != None:
            print(to.display())
        towerOfHanoi(n-1, temp, to, fromrod)
```

The screenshot shows a Windows desktop with a taskbar at the bottom. The taskbar includes the Start button, a search bar, and several application icons. The active window is a Python IDE titled 'pr2.py - G:\harsh1\pr2.py (3.8.4)'. The IDE's menu bar includes 'File', 'Edit', 'Format', 'Run', 'Options', 'Window', and 'Help'. The code editor displays a Python script. It defines a 'Stack' class with methods for initialization, pushing, popping, getting the head, and displaying the stack. Below the class definition, three Stack objects (A, B, C) are created. A recursive function 'towerOfHanoi' is defined, which moves disks between three rods (fromrod, to, temp) according to the Tower of Hanoi algorithm. The code is currently at line 12, column 23.

```
pr2.py - G:\harshl\pr2.py (3.8.4)
File Edit Format Run Options Window Help

class Stack:
    def __init__(self):
        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)
```

Output.

```
Python 3.8.4 Shell
File Edit Shell Debug Options Window Help
Python 3.8.4 [tags/v3.8.4:dfe645a, Jul 13 2020, 16:30:28] [MSC v.1926 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: G:\harshl\pr2.py =====
Enter the number of the disk in rod A : 4
['disk 1']
['disk 2']
['disk 2', 'disk 1']
['disk 3']
['disk 1', 'disk 1']
['disk 3', 'disk 2']
['disk 3', 'disk 2', 'disk 1']
['disk 4']
['disk 4', 'disk 1']
['disk 2']
['disk 2', 'disk 1']
['disk 4', 'disk 3']
['disk 1']
['disk 4', 'disk 3', 'disk 2']
['disk 4', 'disk 3', 'disk 2', 'disk 1']
>>> |
```

## Practical

no:3(c)

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

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.

**Code.**



```
pr2.py - G:\harsh1\pr2.py (3.8.4)
File Edit Format Run Options Window Help

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 add_polynomial(my_list, my_list2):
```

```
pr2.py - G:\harsh1\pr2.py (3.8.4)
File Edit Format Run Options Window Help

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):
    #last_object = self.get_tail()
    #last_object.next = Node(e)
    #self.size += 1

Ln: 206 Col: 0
```

Output.

```
===== RESTART: G:\harsh1\pr2.py =====
Enter the order for polynomial : 2
Enter coefficient for power 2 : 3
Enter coefficient for power 1 : 4
Enter coefficient for power 0 : 5
Enter coefficient for power 2 : 6
Enter coefficient for power 1 : 5
Enter coefficient for power 0 : 6
9
9
11
>>> |

Ln: 33 Col: 4
```

## Practical

no:3(d)

**Aim:** Write a Program to calculate factorial and to compute the factors of a given number.

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 vice versa). Here is the math-like definition of recursion (again):

$\text{factorial}(0) = 1$   $\text{factorial}(N) = N *$

$\text{factorial}(N-1)$

Code.

```
pr2.py - G:\harsh1\pr2.py (3.8.4)
File Edit Format Run Options Window Help

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)
```

Output.

```
Python 3.8.4 Shell
File Edit Shell Debug Options Window Help

Python 3.8.4 (tags/v3.8.4:dfa645e, Jul 13 2020, 16:30:28) [MSC v.1926 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: G:\harsh1\pr2.py =====
Enter Number: 5
Factorial is : 120
Factors of the given numbers is : [1, 5]
Enter number : 4
Factorial is : 24
Factors of the given numbers is : [1, 2, 4]
>>>
```

## 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

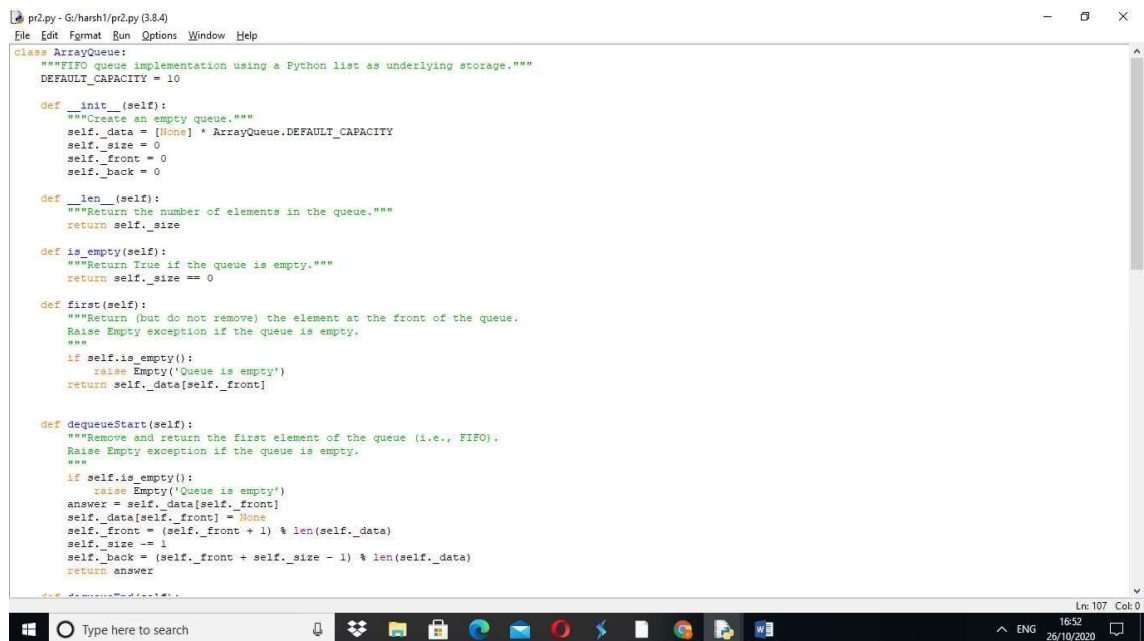
## 1. Enqueue

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

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.



```
pr2.py - G/harsh1/pr2.py (3.8.4)
File Edit Format Run Options Window Help

class ArrayQueue:
    """FIFO queue implementation using a Python list as underlying storage."""
    DEFAULT_CAPACITY = 10

    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
        self._front = (self._front + 1) % len(self._data)
        self._size -= 1
        self._back = (self._front + self._size - 1) % len(self._data)
        return answer
```

Ln: 107 Col: 0

```
pr2.py - G:\harshl\pr2.py (3.8.4)
File Edit Format Run Options Window Help

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
    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))
    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))
    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):
    """Resize to a new list of capacity >= len(self)."""
    old = self._data
    self._data = [None] * cap
    walk = self._front
    for k in range(self._size):
        self._data[k] = old[walk]
        walk = (1 + walk) % len(old)
    self._front = 0
    self._back = 0
    self._size = 0

Ln: 107 Col: 0
```

Output.

```
===== RESTART: G:\harshl\pr2.py =====
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
>>> |

Ln: 22 Col: 4
```

## Practical no: 5

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

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

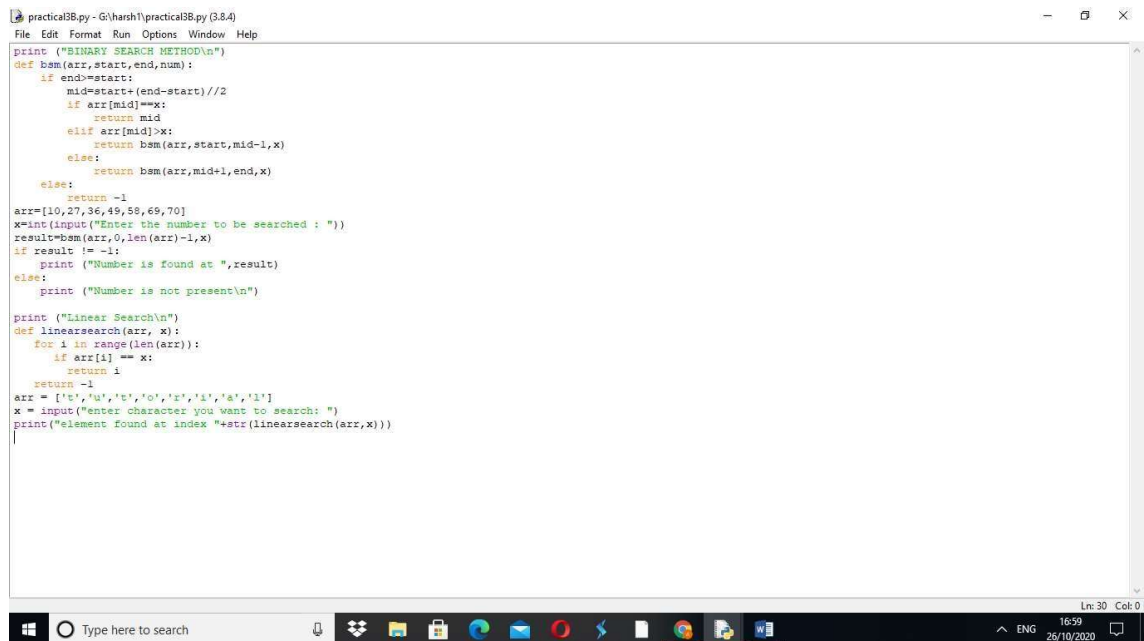
LinearSearch(array, key) for each item in the array if item == value  
return its index.

## 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.

A screenshot of a Python IDE window titled 'practical3B.py - G:\harsh\practical3B.py (3.8.4)'. The code implements two search algorithms. The first is a Binary Search Method (BSM) function that takes an array, start, end, and a number to search. It uses a recursive approach to find the element's index or returns -1 if not found. The second is a Linear Search function that iterates through the array to find the element's index. The code includes test cases for both functions.

```
practical3B.py - G:\harsh\practical3B.py (3.8.4)
File Edit Format Run Options Window Help

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

print ("Linear Search\n")
def linearsearch(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i
    return -1
arr = ['t','u','t','o','r','i','a','l']
x = input("Enter character you want to search: ")
print("element found at index "+str(linearsearch(arr,x)))
```

## Output.

```
>>>===== RESTART: G:\harshl\practical3B.py =====<br>BINARY SEARCH METHOD<br>Enter the number to be searched : 3<br>Number is not present<br><br>Linear Search<br>enter character you want to search: 4<br>element found at index -1<br>>>|</pre><img alt="Screenshot of a Windows command prompt window showing a Python script execution. The script has two search methods: BINARY SEARCH METHOD and Linear Search. The user entered 3 for the binary search, and the output was 'Number is not present'. Then, the user entered 4 for the linear search, and the output was 'element found at index -1'. The Windows taskbar is visible at the bottom with the search bar and various application icons." data-bbox="182 82 881 184"/>
```

## 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.

### Selection sort:

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparisonbased 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.

### Bubble sort:

Bubble sort is a simple sorting algorithm. This sorting algorithm is comparisonbased 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.

### 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.

## Code.

```
pract05.py - G:\harsh\pract05.py (3.8.4)
File Edit Format Run Options Window Help

class Sorting:

    def __init__(self, lst):
        self.lst = lst

    def bubble_sort(self, lst):
        for i in range(len(lst)):
            for j in range(len(lst)):
                if lst[i] < lst[j]:
                    lst[i], lst[j] = lst[j], lst[i]
                else:
                    pass
            return lst

    def selection_sort(self, lst):
        for i in range(len(lst)):
            smallest_element = i
            for j in range(i+1, len(lst)):
                if lst[smallest_element] > lst[j]:
                    smallest_element = j
            lst[i], lst[smallest_element] = lst[smallest_element], lst[i]
            return lst

    def insertion_sort(self, lst):
        for i in range(1, len(lst)):
            index = lst[i]
            j = i-1
            while j >= 0 and index < lst[j]:
                lst[j+1] = lst[j]
                j -= 1
            lst[j+1] = index
            return lst

    def run_sort(self):
        while True:
            print('Select the sorting algorithm:')
            print('1. Bubble Sort.')
            print('2. Selection Sort.')
            print('3. Insertion Sort.')
            print('4. Quit')
            opt = int(input('Option: '))
            if opt == 1:
                print(sort.bubble_sort(self.lst))
            elif opt == 2:
                print(sort.selection_sort(self.lst))
            elif opt == 3:
                print(sort.insertion_sort(self.lst))
            else:
                break
lst = [4, 2, 3, 11, 9, 12, 1, 45, ]
sort = Sorting(lst)
sort.run_sort()
```

```
pract05.py - G:\harsh\pract05.py (3.8.4)
File Edit Format Run Options Window Help

    return lst

    def selection_sort(self, lst):
        for i in range(len(lst)):
            smallest_element = i
            for j in range(i+1, len(lst)):
                if lst[smallest_element] > lst[j]:
                    smallest_element = j
            lst[i], lst[smallest_element] = lst[smallest_element], lst[i]
            return lst

    def insertion_sort(self, lst):
        for i in range(1, len(lst)):
            index = lst[i]
            j = i-1
            while j >= 0 and index < lst[j]:
                lst[j+1] = lst[j]
                j -= 1
            lst[j+1] = index
            return lst

    def run_sort(self):
        while True:
            print('Select the sorting algorithm:')
            print('1. Bubble Sort.')
            print('2. Selection Sort.')
            print('3. Insertion Sort.')
            print('4. Quit')
            opt = int(input('Option: '))
            if opt == 1:
                print(sort.bubble_sort(self.lst))
            elif opt == 2:
                print(sort.selection_sort(self.lst))
            elif opt == 3:
                print(sort.insertion_sort(self.lst))
            else:
                break
lst = [4, 2, 3, 11, 9, 12, 1, 45, ]
sort = Sorting(lst)
sort.run_sort()
```

## Output.

```
===== RESTART: G:\harshl\pract6ds.py =====
Select the sorting algorithm:
1. Bubble Sort.
2. Selection Sort.
3. Insertion Sort.
4. Quit
Option: 1
[1, 2, 3, 4, 5, 11, 12, 45]
Select the sorting algorithm:
1. Bubble Sort.
2. Selection Sort.
3. Insertion Sort.
4. Quit
Option: 3
[1, 2, 3, 4, 5, 11, 12, 45]
Select the sorting algorithm:
1. Bubble Sort.
2. Selection Sort.
3. Insertion Sort.
4. Quit
Option: 2
[1, 2, 3, 4, 5, 11, 12, 45]
Select the sorting algorithm:
1. Bubble Sort.
2. Selection Sort.
3. Insertion Sort.
4. Quit
Option: 4
>>> |
```

## Practical no:7(a)

**Aim:** Write a program to implement the collision technique.

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.

## 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

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:

```
pract6ds.py - G:\harshi\pract6ds.py (3.8.4)
File Edit Format Run Options Window Help

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_index += 1
        list_of_list_index[list_index] = value
    print("After : " + str(list_of_list_index))
```

Output:

```
>>>
-----RESTART: G:\harshi\pract6ds.py-----
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 no:7(b)

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

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.

```
pr2.py - G:/harsh1/pr2.py (3.8.4)
File Edit Format Run Options Window Help

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("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

Ln: 30 Col: 0
```

```
pr2.py - G:/harsh1/pr2.py (3.8.4)
File Edit Format Run Options Window Help

    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("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))

Ln: 30 Col: 0
```

Output:

```
>>>
===== RESTART: G:/harsh1/pr2.py =====
Before : [None, None, None, None]
hash value for 23 is : 3
hash value for 43 is : 3
hash value for 1 is : 1
hash value for 87 is : 3
After: [None, None, None, 87]
>>>

Ln: 127 Col: 4
```

## Practical no. (8)



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

## Tree traversal:

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.

## Uses of Inorder

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

Inorder traversal where Inorder traversal is reversed can be used.

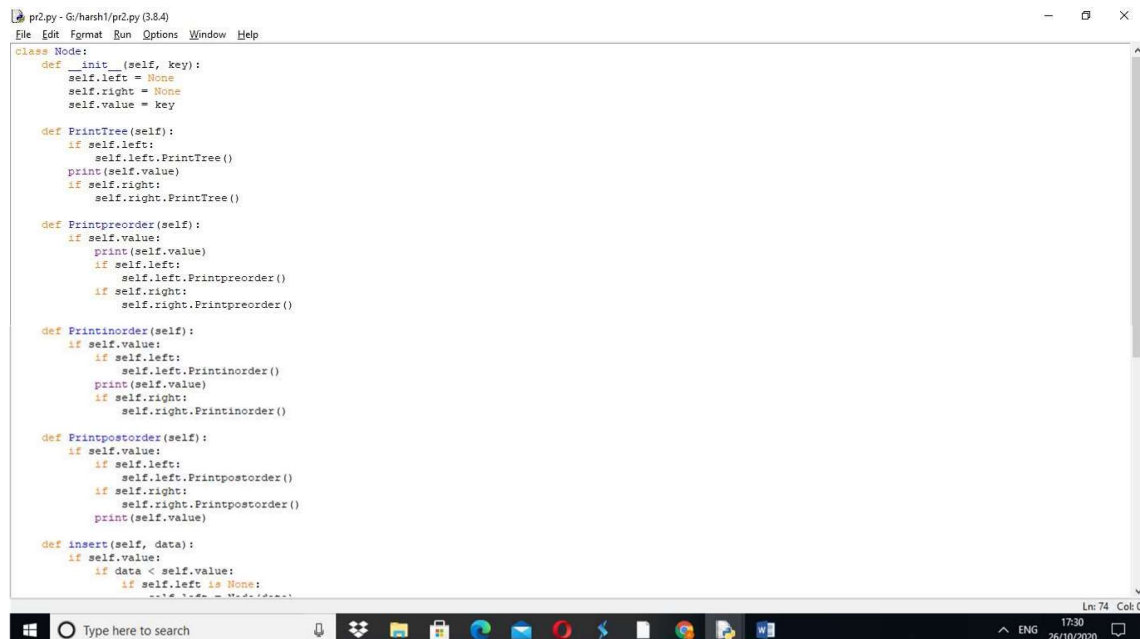
## Uses of Preorder

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

## 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.

## Code.

A screenshot of a Windows-style code editor window titled 'pr2.py - G:\harshi\pr2.py (3.8.4)'. The window contains a Python class named 'Node' with several methods for binary tree traversal. The methods include: 'init' to initialize a node with a key; 'PrintTree' for a recursive in-order traversal; 'Printpreorder' for a recursive pre-order traversal; 'Printinorder' for a recursive in-order traversal (repeated in the code); 'Printpostorder' for a recursive post-order traversal; and 'insert' to add a new node to the tree based on its value. The code is written in Python 3.8.4 syntax. The Windows taskbar at the bottom shows the search bar, taskbar icons, and system clock indicating 17:30 on 26/10/2020.

```
pr2.py - G:\harshi\pr2.py (3.8.4)
File Edit Format Run Options Window Help

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)
            else:
                if self.right is None:
                    self.right = Node(data)
                else:
                    self.right.insert(data)
```

```
pr2.py - G:/harsh1/pr2.py (3.8.4)
File Edit Format Run Options Window Help
def __init__(self, data):
    self.data = data
    self.left = None
    self.right = None
    self.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_l = Node(None)
    root_l.insert(28)
    root_l.insert(4)
    root_l.insert(13)
    root_l.insert(130)
    root_l.insert(123)
    print("Now ordering with insert")
    root_l.PrintTree()
    print("Pre order")
    root_l.Printpreorder()
    print("In Order")
    root_l.Printinorder()
    print("Post Order")
    root_l.Printpostorder()
```

## Output.

```
===== RESTART: G:/harsh1/pr2.py =====
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
130
28
>>> |
```

