



**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./Ms. Abhay Jha

Roll No: 321

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)

**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.

**Github Repository Link :**

<https://github.com/AJ-Walker/DS/blob/master/practical1a.py>

### **Theory :**

**Searching:** You can search an array for a certain value, and return the indexes that get a match. To search an array, use the `search()` method.

**Sorting:** Sorting means putting elements in an ordered sequence. Ordered sequence is any sequence that has an order corresponding to elements, like numeric or alphabetical, ascending or descending. The NumPy ndarray object has a function called `sort()`, that will sort a specified array.

**Merging:** There are several ways to join, or concatenate, two or more lists in Python. One of the easiest ways are by using the `+` operator. Another way to join two lists are by appending all the items from list2 into list1, one by one or you can use the `extend()` method, which purpose is to add elements from one list to another list.

**Reversing:** Using the `reversed()` built-in function.

In this method, we neither reverse a list in-place(modify the original list), nor we create any copy of the list. Instead, we get a reverse iterator which we use to cycle through the list.

Using the `reverse()` built-in function.

Using the `reverse()` method we can reverse the contents of the list object in-place i.e., we don't need to create a new list instead we just copy the existing elements to the original list in reverse order. This method directly modifies the original list.

## Code :

```
practical1a.py - C:\Users\user\abhay\python\DataStructures\practical1a.py (3.8.1)
File Edit Format Run Options Window Help

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

class ArrayOperations:
    def __init__(self, arr):
        self.array = arr
    def search(self, e):
        if e in self.array:
            return True
        return False

    def sort(self):
        for i in range(len(self.array)):
            lowest_index = i
            for j in range(i+1, len(self.array)):
                if self.array[j] < self.array[lowest_index]:
                    lowest_index = j
            self.array[i], self.array[lowest_index] = self.array[lowest_index], self.array[i]
        return self.array

    def merge_el(self, l):
        self.array = self.array + l
        return self.array

    def reverse(self):
        return self.array[::-1]

arr = [34,67,12,89,23,90]
array1 = ArrayOperations(arr)
print(array1.sort())
print(array1.search(89))
print(array1.merge_el([56,43,21,87]))
print(array1.reverse())
```

## Output :

```
Python 3.8.1 Shell
File Edit Shell Debug Options Window Help

Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 23:11:46) [MSC v.1916 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Users\user\abhay\python\DataStructures\practical1a.py =====
[12, 23, 34, 67, 89, 90]
True
[12, 23, 34, 67, 89, 90, 56, 43, 21, 87]
[87, 21, 43, 56, 90, 89, 67, 34, 23, 12]
>>> |
```

## **Practical No. 1.B**

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

**Github Repository Link :**

<https://github.com/AJ-Walker/DS/blob/master/Practical1b.py>

### **Theory :**

**Matrix Addition:** In Python, we can implement a matrix as a nested list (list inside a list). We can treat each element as a row of the matrix.

For example  $X = [[1, 2], [4, 5], [3, 6]]$  would represent a 3x2 matrix. First row can be selected as  $X[0]$  and the element in first row, first column can be selected as  $X[0][0]$ .

We can perform matrix addition in various ways in Python:

Matrix Addition using Nested Loop.

Matrix Addition using Nested List Comprehension.

**Matrix Multiplication:** In Python, we can implement a matrix as nested list (list inside a list). We can treat each element as a row of the matrix.

For example  $X = [[1, 2], [4, 5], [3, 6]]$  would represent a 3x2 matrix.

The first row can be selected as  $X[0]$ . And, the element in first row, first column can be selected as  $X[0][0]$ .

Multiplication of two matrices X and Y is defined only if the number of columns in X is equal to the number of rows Y.

If X is a  $n \times m$  matrix and Y is a  $m \times l$  matrix then, XY is defined and has the dimension  $n \times l$  (but YX is not defined). Here are a couple of ways to implement matrix multiplication in Python.

We can perform matrix multiplication in various ways in Python:

Matrix Multiplication using Nested Loop.

Matrix Multiplication using Nested List Comprehension.

**Matrix Transpose:** In Python, we can implement a matrix as a nested list (list inside a list). We can treat each element as a row of the matrix. For example  $X = [[1, 2], [4, 5],$

[3, 6]] would represent a 3x2 matrix. The first row can be selected as X[0]. And, the element in the first-row first column can be selected as X[0][0].

Transpose of a matrix is the interchanging of rows and columns. It is denoted as X'. The element at ith row and jth column in X will be placed at jth row and ith column in X'. So if X is a 3x2 matrix, X' will be a 2x3 matrix.

We can perform matrix transpose various ways in Python:

Matrix Transpose using Nested Loop.

Matrix Transpose using Nested List Comprehension.

### Code :

```
# Write a program to perform the Matrix addition, Multiplication, and
# Transpose Operation
a=str(input("Enter the * for Multiplication Operation , + Adding Operation ,T for Transpose Operation :"))
X = [[12,7,3],
      [4 ,5,6],
      [7 ,8,9]]

Y = [[5,8,1],
      [6,7,3],
      [4,5,9]]

result = [[0,0,0],
          [0,0,0],
          [0,0,0]]

if a== '+':
    print("Adding Operation")
    def Adding_Matrix(X,Y):
        for i in range(len(X)):
            for j in range(len(X[0])):
                result[i][j] = X[i][j] + Y[i][j]

        for r in result:
            print(r)

    Adding_Matrix(X,Y)
```

```
elif a == "t" or a == "T":
    print("Transpose Operation")
    for i in range(len(X)):
        for j in range(len(X[0])):
            result[j][i] = X[i][j]

    for r in result:
        print(r)

else:
    print("Enter valid input:")
```

## Output :

```
Enter the * for Multiplication Operation , + Adding Operation ,T for Transpose Operation :*
Multiplication Operation
[114, 160, 60]
[74, 97, 73]
[119, 157, 112]
>>>
===== RESTART: C:\Users\user\abhay\python\DataStructures\Practical1b.py =====
Enter the * for Multiplication Operation , + Adding Operation ,T for Transpose Operation :+
Adding Operation
[17, 15, 4]
[10, 12, 9]
[11, 13, 18]
>>>
===== RESTART: C:\Users\user\abhay\python\DataStructures\Practical1b.py =====
Enter the * for Multiplication Operation , + Adding Operation ,T for Transpose Operation :T
Transpose Operation
[12, 4, 7]
[7, 5, 8]
[3, 6, 9]
```

## **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

**Github Repository Link :**

<https://github.com/AJ-Walker/DS/blob/master/practical2.py>

### **Theory :**

Inserting element in the linked list involves reassigning the pointers from the existing nodes to the newly inserted node. Depending on whether the new data element is getting inserted at the beginning or at the middle or at the end of the linked list, we have the below scenarios.

#### **Inserting at the Beginning of the Linked List**

This involves pointing the next pointer of the new data node to the current head of the linked list. So the current head of the linked list becomes the second data element and the new node becomes the head of the linked list.

#### **Inserting at the End of the Linked List**

This involves pointing the next pointer of the the current last node of the linked list to the new data node. So the current last node of the linked list becomes the second last data node and the new node becomes the last node of the linked list.

We can remove an existing node using the key for that node. In the below program we locate the previous node of the node which is to be deleted. Then point the next pointer of this node to the next node of the node to be deleted.

Singly linked lists can be traversed in only forward direction starting from the first data element. We simply print the value of the next data element by assigning the pointer of the next node to the current data element.



## Code :

\*practical2.py - C:\Users\user\abhay\python\DataStructures\practical2.py (3.8.1)\*

File Edit Format Run Options Window Help

```
#Implement Linked List. Include options for insertion, deletion, and
# search of a number,
# reverse the list and concatenate two linked lists
class Node:
    def __init__(self, data, next = None):
        self.data = data
        self.next = None

class LinkedList():
    def __init__(self):
        self.head = None

    def printList(self):
        temp = self.head
        while(temp):
            print(temp.data, end=" ")
            temp = temp.next

    def insertion(self, data):
        new_node = Node(data)
        if self.head is None:
            self.head = new_node
            return

        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node

    def delete_node(self, key):
        current_node = self.head
        if current_node and current_node.data == key:
            self.head = current_node.next
            current_node = None
            return
```

File Edit Format Run Options Window Help

```
prev = None
while current_node and current_node.data != key:
    prev = current_node
    current_node = current_node.next
if current_node:
    prev.next = current_node.next
    current_node = None
    return
else:
    return

def length(self):
    current_node = self.head
    count = 0
    while current_node:
        current_node = current_node.next
        count += 1
    return count

def reverse(self):
    prev = None
    current = self.head
    while(current is not None):
        next = current.next
        current.next = prev
        prev = current
        current = next
    self.head = prev

def search_list(self, data):
    count = 0
    pointer = self.head

    while pointer:
        if pointer.data == data:
```

```
        pointer.data == data:
            count += 1
        pointer = pointer.next
    print(f"\nNumber {data} is there in list")
    return count

def mergeLists(List_1, List_2):
    head_ptr = temp_ptr = Node(List_1)
    while List_1 or List_2:
        if List_1 and (not List_2 or List_1.data <= List_2.data):
            temp_ptr.next = Node(List_1.data)
            List_1 = List_1.next
        else:
            temp_ptr.next = Node(List_2.data)
            List_2 = List_2.next
        temp_ptr = temp_ptr.next
    return head_ptr.next

l1 = LinkedList()
l1 = LinkedList()
l1.insertion(56)
l1.insertion(5)
l1.insertion(34)
l1.insertion(21)
l1.insertion(78)
l1.insertion(59)
print("Linked List : ")
l1.printList()
l1.search_list(5)
print("Deleting...")
l1.delete_node(34)
print("After deleting : ")
l1.printList()
print("\nReversing the list : ")
l1.reverse()
l1.printList()
print("\nMerging two linked list : ")
l2 = LinkedList()
l2.head = mergeLists(l1.head, l1.head)
l2.printList()
```

## Output :

```
===== RESTART: C:\Users\user\abhay\python\DataStructures\practical
Linked List :
56 5 34
Number 5 is there in list
Deleting...
After deleting :
56 5
Reversing the list :
5 56
Merging two linked list :
5 21 56 78 59
>>> |
```

### **Practical No. 3.A**

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

**Github Repository Link :**

<https://github.com/AJ-Walker/DS/blob/master/Practical3.a.py>

#### **Theory :**

A stack data structure can be implemented using a one-dimensional array. But stack implemented using array stores only a fixed number of data values. Just define a one dimensional array of specific size and insert or delete the values into that array by using **LIFO principle** with the help of a variable called '**top**'. Initially, the top is set to -1. Whenever we want to insert a value into the stack, increment the top value by one and then insert. Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one.

push(value) - Inserting value into the stack

In a stack, push() is a function used to insert an element into the stack. In a stack, the new element is always inserted at top position. Push function takes one integer value as parameter and inserts that value into the stack. We can use the following steps to push an element on to the stack...

**Step 1** - Check whether stack is FULL. (top == SIZE-1)

**Step 2** - If it is FULL, then display "Stack is FULL!!! Insertion is not possible!!!" and terminate the function.

**Step 3** - If it is NOT FULL, then increment top value by one (top++) and set stack[top] to value (stack[top] = value)

pop() - Delete a value from the Stack

In a stack, pop() is a function used to delete an element from the stack. In a stack, the element is always deleted from top position. Pop function does not take any value as parameter. We can use the following steps to pop an element from the stack...

**Step 1** - Check whether stack is EMPTY. (top == -1)

**Step 2** - If it is EMPTY, then display "Stack is EMPTY!!! Deletion is not possible!!!" and terminate the function

Step 3 - If it is NOT EMPTY, then delete stack[top] and decrement top value by one (top).

display() - Displays the elements of a Stack

We can use the following steps to display the elements of a stack...

Step 1 - Check whether stack is EMPTY. (top == -1)

Step 2 - If it is EMPTY, then display "Stack is EMPTY!!!" and terminate the function.

Step 3 - If it is NOT EMPTY, then define a variable 'i' and initialize with top. Display stack[i] value and decrement i value by one (i--).

Step 4 - Repeat above step until i value becomes '0'.

### Code :

```
Practical3.a.py - C:\Users\user\abhay\python\DataStructures\Practical3.a.py (3.8.1)
File Edit Format Run Options Window Help

# Implement the following for Stack:
# Perform Stack operations using Array implementation.
class Stack:
    def __init__(self,data=[]):
        self.data = []

    def is_empty(self):
        self.len_data = len(self.data)
        return self.len_data

    def push(self,element):
        self.data.append(element)

    def pop(self):
        if self.is_empty():
            print("Array is empty")
        else:
            self.data.pop()

    def top_element(self):
        print("Top element of stack : ",self.data[-1])

    def display(self):
        print(self.data)
        print(self.is_empty())

stackOperation = Stack()
stackOperation.display()
stackOperation.push(23)
stackOperation.push(45)
stackOperation.push(12)
stackOperation.display()
stackOperation.top_element()
stackOperation.pop()
stackOperation.display()
```

## Output :

```
Python 3.8.1 Shell
File Edit Shell Debug Options Window Help
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 23:11:46) [MSC v.1916 64 bit (AMD64)]
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Users\user\abhay\python\DataStructures\Practical3.a.py =====
[]
0
[23, 45, 12]
3
Top element of stack : 12
Array is empty
[23, 45, 12]
3
Array is empty
Array is empty
[23, 45, 12]
3
Array is empty
>>>|
```

## **Practical No. 3.B**

**Aim :** Implement Tower of Hanoi.

**Github Repository Link :**

<https://github.com/AJ-Walker/DS/blob/master/Practical3.b.py>

**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:

Only one disk can be moved at a time.

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.

No disk may be placed on top of a smaller disk.

Note: Transferring the top n-1 disks from source rod to Auxiliary rod can again be thought of as a fresh problem and can be solved in the same manner.

**Code :**

```
File Edit Format Run Options Window Help
def hanoi(n, P1, P2, P3):
    if n == 0:
        return

    global count
    count += 1
    hanoi(n-1, P1, P3, P2)

    if P1:
        P3.append(P1.pop())
        print(f"Pole 1 : {A}, Pole 2 : {B}, Pole 3 : {C}\n")

    hanoi(n-1, P2, P1, P3)

n = 3
A = list(range(n,0,-1))
B, C = [], []
print(f"{A[0]} is big disc, {A[1]} is medium disc, {A[2]} is small disc.")
print(f"Pole 1 : {A}, Pole 2 : {B}, Pole 3 : {C}\n")
count = 0
hanoi(n, A, B, C)
print(f"Total Steps : {count}")
```



**Output :**

3 is big disc, 2 is medium disc, 1 is small disc.

Pole 1 : [3, 2, 1], Pole 2 : [], Pole 3 : []

Pole 1 : [3, 2], Pole 2 : [], Pole 3 : [1]

Pole 1 : [3], Pole 2 : [2], Pole 3 : [1]

Pole 1 : [3], Pole 2 : [2, 1], Pole 3 : []

Pole 1 : [], Pole 2 : [2, 1], Pole 3 : [3]

Pole 1 : [1], Pole 2 : [2], Pole 3 : [3]

Pole 1 : [1], Pole 2 : [], Pole 3 : [3, 2]

Pole 1 : [], Pole 2 : [], Pole 3 : [3, 2, 1]

Total Steps : 7

>>> |

### **Practical No. 3.C**

**Aim :** WAP to scan a polynomial using linked list and add two polynomial.

**Github Repository Link :**

<https://github.com/AJ-Walker/DS/blob/master/Practical3.c.py>

#### **Theory :**

A linked list is a sequence of data elements, which are connected together via links. Each data element contains a connection to another data element in form of a pointer. Python does not have linked lists in its standard library. We implement the concept of linked lists using the concept of nodes as discussed in the previous chapter. We have already seen how we create a node class and how to traverse the elements of a node. In this chapter we are going to study the types of linked lists known as singly linked lists. In this type of data structure there is only one link between any two data elements. We create such a list and create additional methods to insert, update and remove elements from the list



## Code :

Practical3.c.py - C:\Users\user\abhay\python\DataStructures\Practical3.c.py (3.8.1)

File Edit Format Run Options Window Help

```
# Implement the following for Stack:  
# WAP to scan a polynomial using linked list and add two polynomial.
```

```
class PolyNode(object):  
    def __init__(self, degree, coefficient):  
        self.degree = degree  
        self.coefficient = coefficient  
        self.next = None  
  
class Polynomial:  
    def __init__(self, degree=None, coefficient=None):  
        if degree is None:  
            self.polyHead = None  
        else:  
            self.polyHead = PolyNode(degree, coefficient)  
            self.polyTail = self.polyHead  
  
    def degree(self):  
        if self.polyHead is None:  
            return -1  
        else:  
            return self.polyHead.degree  
  
    def __add__(self, rhsPoly):  
        assert self.degree() >= 0 and rhsPoly.degree() >= 0  
        newPoly = Polynomial()  
        nodeA = self.polyHead  
        nodeB = rhsPoly.polyHead
```

```
while nodeA is not None and nodeB is not None:
```

```
    if nodeA.degree > nodeB.degree:
```

```
        degree = nodeA.degree
```

```
        coefficient = nodeA.coefficient
```

```
        nodeA = nodeA.next
```

```
    elif nodeA.degree < nodeB.degree:
```

```
        degree = nodeB.degree
```

```
        coefficient = nodeB.coefficient
```

```
        nodeB = nodeB.next
```

```
    else:
```

```
        degree = nodeA.degree
```

```
        coefficient = nodeA.coefficient + nodeB.coefficient
```

```
        nodeA = nodeA.next
```

```
        nodeB = nodeB.next
```

```
newPoly.appendTerm(degree, coefficient)
```

```
while nodeA is not None:
```

```
    newPoly.appendTerm(nodeA.degree, nodeA.coefficient)
```

```
    nodeA = nodeA.next
```

```
while nodeB is not None:
```

```
    newPoly.appendTerm(nodeB.degree, nodeB.coefficient)
```

```
    nodeB = nodeB.next
```

```
return newPoly
```

```
def appendTerm(self, degree, coefficient):
```

```
    if coefficient != 0.0:
```

```
        newTerm = PolyNode(degree, coefficient)
```

```
        if self.polyHead is None:
            self.polyHead = newTerm
        else:
            self.polyTail.next = newTerm

        self.polyTail = newTerm

    def printPoly(self):
        currentNode = self.polyHead
        while currentNode is not None:
            if currentNode.next is not None:
                print(f"{currentNode.coefficient}x^{currentNode.degree} + ")
            else:
                print(f"{currentNode.coefficient}x^{currentNode.degree}")
            currentNode = currentNode.next

if __name__ == "__main__":
    poly1 = Polynomial(2, 5)
    poly1 += Polynomial(1, 3)
    poly1 += Polynomial(0, -10)
    print("First Polynomial : ")
    poly1.printPoly()
    poly2 = Polynomial(3, 2)
    poly2 += Polynomial(2, 4)
    poly2 += Polynomial(0, 3)
    print("Second Polynomial : ")
    poly2.printPoly()
    addPoly = poly1 + poly2
    print("The addition of the two polynomials is:")
    addPoly.printPoly()
```

## Output :

```
===== RESTART: C:\Users\user\abhay\python\DataStructures\Practical3.c.py :
First Polynomial :
5x^2 +
3x^1 +
-10x^0
Second Polynomial :
2x^3 +
4x^2 +
3x^0
The addition of the two polynomials is:
2x^3 +
9x^2 +
3x^1 +
-7x^0
>>> |
```

### **Practical No. 3.D**

**Aim :** WAP to calculate factorial and to compute the factors of a given no. (i) using recursion, (ii) using iteration.

**Github Repository Link :**

<https://github.com/AJ-Walker/DS/blob/master/Practical3.d.py>

**Theory :**

Factorial of Numbers:

The factorial of a number is the product of all the integers from 1 to that number.

For example, the factorial of 6 is  $1*2*3*4*5*6 = 720$ . Factorial is not defined for negative numbers and the factorial of zero is one,  $0! = 1$ .

Iterative Solution:

Factorial can also be calculated iteratively . Here we have shown the iterative approach using both for and while loop.

Recursive Solution:

Factorial can also be calculated recursively. Recursive is calling the function inside of a function.

Factors of the Numbers:

In order to find factors of a number, we have to run a loop over all numbers from 1 to itself and see if it is divisible. If i is able to divide num completely, it is added in the list. Finally the list is displayed as the factors of given number

Iterative Solution:

Factors can also be calculated iteratively . Here we have shown the iterative approach using both for and while loop.

Recursive Solution:

Factorial can also be calculated recursively. Recursive is calling the function inside of a function.

**Code :**

Practical3.d.py - C:\Users\user\abhay\python\DataStructures\Practical3.d.py (3.8.1)

File Edit Format Run Options Window Help

```
# Implement the following for Stack:  
# WAP to calculate factorial and to compute the factors of a given no.  
# (i) using recursion, (ii) using iteration.
```

```
def factorial_num(num):  
    fact = 1  
    while (num>0):  
        fact = fact * num  
        num = num - 1  
    return fact  
  
def factorial_num_recursion(num):  
    if num == 1:  
        return num  
    else :  
        return num*factorial_num_recursion(num-1)  
  
def factors_num(num):  
    for i in range(1,num+1):  
        if (num%i)==0:  
            print(i, end = " ")  
  
def factors_num_recursion(num,x):  
    if x <= num:  
        if (num % x == 0):  
            print(x, end = " ")  
        factors_num_recursion(num, x + 1)  
  
num = 6  
print(f"Factorial of {num} : {factorial_num(num)}")  
print(f"Factorial of {num} using recursion : {factorial_num_recursion(num)}")  
print(f"Factors of {num} : ")  
factors_num(num)  
print(f"\nFactors of {num} using recursion : ")  
print(factors_num_recursion(num,1))
```

## Output :

File Edit Shell Debug Options Window Help

Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 23:11:46) [MSC v.1916 64 bit (AMD64)] o  
Type "help", "copyright", "credits" or "license()" for more information.

>>>

===== RESTART: C:\Users\user\abhay\python\DataStructures\Practical3.d.py =====

Factorial of 6 : 720

Factorial of 6 using recursion : 720

Factors of 6 :

1 2 3 6

Factors of 6 using recursion :

1 2 3 6 None

>>>|

## **Practical No. 4**

**Aim :** Perform Queues operations using Circular Array implementation.

**Github Repository Link :**

<https://github.com/AJ-Walker/DS/blob/master/Practical4.py>

### **Theory :**

In the circular queue, the first index comes right after the last index. You can think of a circular queue as shown in the following figure. Circular queue will be full when  $\text{front} = -1$  and  $\text{rear} = \text{max}-1$ . Implementation of circular queue is similar to that of a linear queue. Only the logic part that is implemented in the case of insertion and deletion is different from that in a linear queue.

There are three scenario of inserting an element in a queue.

If  $(\text{rear} + 1) \% \text{maxsize} = \text{front}$ , the queue is full. In that case, overflow occurs and therefore, insertion can not be performed in the queue.

If  $\text{rear} \neq \text{max} - 1$ , then rear will be incremented to the  $\text{mod}(\text{maxsize})$  and the new value will be inserted at the rear end of the queue.

If  $\text{front} \neq 0$  and  $\text{rear} = \text{max} - 1$ , then it means that queue is not full therefore, set the value of rear to 0 and insert the new element there.

To delete an element from the circular queue, we must check for the three following conditions.

If  $\text{front} = -1$ , then there are no elements in the queue and therefore this will be the case of an underflow condition.

If there is only one element in the queue, in this case, the condition  $\text{rear} = \text{front}$  holds and therefore, both are set to -1 and the queue is deleted completely.

If  $\text{front} = \text{max} - 1$  then, the value is deleted from the front end the value of front is set to 0.

Otherwise, the value of front is incremented by 1 and then delete the element at the front end.



## Code :

```
Practical4.py - C:\Users\user\abhay\python\DataStructures\Practical4.py (3.8.1)
File Edit Format Run Options Window Help

# Perform Queues operations using Circular Array implementation

class CircularQueue():
    def __init__(self,size):
        self.size = size
        self.data = [None for i in range(size)]
        self.front = -1
        self.rear = -1

    def enqueue(self, data):
        if ((self.rear + 1) % self.size == self.front):
            print("Queue is Full")
        elif (self.front == -1):
            self.front = 0
            self.rear = 0
            self.data[self.rear] = data
        else:
            self.rear = (self.rear + 1) % self.size
            self.data[self.rear] = data

    def dequeue(self):
        if (self.front == -1):
            print("Queue is Empty")
        elif (self.front == self.rear):
            temp=self.data[self.front]
            self.front = -1
            self.rear = -1
            return temp
        else:
            temp = self.data[self.front]
            self.front = (self.front + 1) % self.size
            return temp

    def display(self):
        if(self.front == -1):
            print("Queue is Empty")
        elif (self.rear >= self.front):
            print("Elements in the circular queue are:",end=" ")
            for i in range(self.front, self.rear + 1):
                print(self.data[i], end = " ")
            print()
        else:
            print("Elements in Circular Queue are:",end=" ")
            for i in range(self.front, self.data):
                print(self.data[i], end = " ")
            for i in range(0, self.rear + 1):
                print(self.data[i], end = " ")
            print()

        if ((self.rear + 1) % self.size == self.front):
            print("Queue is Full")

cq = CircularQueue(5)
cq.enqueue(23)
cq.enqueue(34)
cq.enqueue(12)
cq.enqueue(12)
cq.enqueue(67)
cq.enqueue(78)
cq.display()
cq.dequeue()
cq.display()
```

## Output :

```
Type "help", "copyright", "credits" or "license()" for more information.
>>>
===== RESTART: C:\Users\user\abhay\python\DataStructures\Practical4.py ==
Queue is Full
Elements in the circular queue are: 23 34 12 12 67 Queue is Full
Elements in the circular queue are: 34 12 12 67
>>>
```

## **Practical No. 5**

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

**Github Repository Link :**

<https://github.com/AJ-Walker/DS/blob/master/practical5.py>

### **Theory :**

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.

Linear Search:

In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data structure.

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.

**Code :**

practical5.py - C:\Users\user\abhay\python\DataStructures\practical5.py (3.8.1)

File Edit Format Run Options Window Help

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

```
def LinearSearch(array, element_l):  
    for i in range(len(array)):  
        if array[i] == element_l:  
            return i  
    return -1
```

```
def BinarySearch(array, element_l):  
    first = 0  
    array.sort()  
    last = len(array)-1  
    done = False  
    while (first <= last) and not done:  
        mid = (first+last)//2  
        if array[mid] == element_l:  
            done = True  
        else:  
            if element_l < array[mid]:  
                last = last - 1  
            else:  
                first = first + 1  
    return done
```

```
array = [45,78,23,17,453,7]
```

```
print(array)
```

```
element_l = int(input("Enter the element you want to search : "))
```

```
print("Select 'L' for Linear Search and 'B' for Binary Search")
```

```
yourChoice = str(input("Enter your choice : "))
```

```
if yourChoice == 'L':
```

```
    result = LinearSearch(array, element_l)
```

```
    print(result)
```

```
    if result == 1:
```

```
        print("Element is present.")
```

```
    else :
```

```
        print("Element not present.")
```

```
else:
```

```
    result = BinarySearch(array, element_l)
```

```
    print(result)
```

```
    if result == -1:
```

```
        print("Element not present.")
```

```
    else :
```

```
        print("Element is present.")
```



### Output :

```
>>>
```

```
===== RESTART: C:\Users\user\abhay\python\DataStructures\practical5.py ==
```

```
[45, 78, 23, 17, 453, 7]
```

```
Enter the element you want to search : 17
```

```
Select 'L' for Linear Search and 'B' for Binary Search
```

```
Enter your choice : L
```

```
3
```

```
Element is present.
```

```
>>>
```

```
===== RESTART: C:\Users\user\abhay\python\DataStructures\practical5.py ==
```

```
[45, 78, 23, 17, 453, 7]
```

```
Enter the element you want to search : 17
```

```
Select 'L' for Linear Search and 'B' for Binary Search
```

```
Enter your choice : B
```

```
True
```

```
Element is present.
```

```
>>>
```

## **Practical No. 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 Repository Link :**

<https://github.com/AJ-Walker/DS/blob/master/practical6.py>

### **Theory :**

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.

#### **Bubble Sort**

It is a comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order.

#### **Insertion Sort**

Insertion sort involves finding the right place for a given element in a sorted list. So in beginning we compare the first two elements and sort them by comparing them. Then we pick the third element and find its proper position among the previous two sorted elements. This way we gradually go on adding more elements to the already sorted list by putting them in their proper position.

#### **Selection Sort**

In selection sort we start by finding the minimum value in a given list and move it to a sorted list. Then we repeat the process for each of the remaining elements in the unsorted list. The next element entering the sorted list is compared with the existing elements and placed at its correct position. So at the end all the elements from the unsorted list are sorted.

## Code :

practical6.py - C:\Users\user\abhay\python\DataStructures\practical6.py (3.8.1)

File Edit Format Run Options Window Help

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

```
def insertion_sort(array):
    for i in range(1, len(array)):
        item = array[i]
        j = i - 1
        while j >= 0 and array[j] > item:
            array[j + 1] = array[j]
            j -= 1
        array[j + 1] = item
    return array

def bubble_sort(array):
    n = len(array)
    for i in range(n):
        already_sorted = True
        for j in range(n - i - 1):
            if array[j] > array[j + 1]:
                array[j], array[j + 1] = array[j + 1], array[j]
                already_sorted = False
        if already_sorted:
            break
    return array

def selection_sort(array):
    for i in range(len(array)):
        minimum = i
        for j in range(i + 1, len(array)):
            if array[j] < array[minimum]:
                minimum = j
        array[minimum], array[i] = array[i], array[minimum]
    return array

array = [45,89,23,98,5,78,32]
print("The list is : ",array)
#array = list(input("Enter your list : "))
print("Select 'I' for Insertion sort, 'B' for Bubble sort, 'S' for selection sort")
yourChoice = str(input("Your choice : "))

if yourChoice == "I":
    print("Insertion sort : ",insertion_sort(array))
elif yourChoice == "B":
    print("Bubble sort : ",bubble_sort(array))
else:
    print("Selection sort : ",selection_sort(array))
```

## Output :

```
===== RESTART: C:\Users\user\abhay\python\DataStructures\practical6.py ===  
The list is : [45, 89, 23, 98, 5, 78, 32]  
Select 'I' for Insertion sort, 'B' for Bubble sort, 'S' for selection sort  
Your choice : B  
Bubble sort : [5, 23, 32, 45, 78, 89, 98]  
>>>  
===== RESTART: C:\Users\user\abhay\python\DataStructures\practical6.py ===  
The list is : [45, 89, 23, 98, 5, 78, 32]  
Select 'I' for Insertion sort, 'B' for Bubble sort, 'S' for selection sort  
Your choice : I  
Insertion sort : [5, 23, 32, 45, 78, 89, 98]  
>>>  
===== RESTART: C:\Users\user\abhay\python\DataStructures\practical6.py ===  
The list is : [45, 89, 23, 98, 5, 78, 32]  
Select 'I' for Insertion sort, 'B' for Bubble sort, 'S' for selection sort  
Your choice : S  
Selection sort : [5, 23, 32, 45, 78, 89, 98]  
>>>
```

## **Practical No. 7.A**

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

**Github Repository Link :**

<https://github.com/AJ-Walker/DS/blob/master/Practical7a.py>

**Theory :**

Hash functions are there to map different keys to unique locations (index in the hash table), and any hash function which is able to do so is known as the perfect hash function. Since the size of the hash table is very less comparatively to the range of keys, the perfect hash function is practically impossible. What happens is, more than one keys map to the same location and this is known as a collision. A good hash function should have less number of collisions. Collision resolution is finding another location to avoid the collision.

The most popular resolution techniques are,

Separate chaining

Open addressing

**Code :**

```
# Implement the following for Hashing:  
# Write a program to implement the collision technique.
```

```
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:  
  
        list_index = Hash(value, 0, len(list_of_keys)).get_key_value()  
        if list_of_list_index[list_index]:  
            print("Collision detected")  
        else:  
            list_of_list_index[list_index] = value  
  
    print("After: " + str(list_of_list_index))
```

### Output :

```
===== RESTART: C:\Users\user\abhay\p  
Before : [None, None, None, None]  
Collision detected  
Collision detected  
After: [None, 1, None, 23]  
>>> |
```

## **Practical No. 7.B**

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

**Github Repository Link :**

<https://github.com/AJ-Walker/DS/blob/master/Practical7b.py>

**Theory :**

Linear probing is a component of open addressing schemes for using a hash table to solve the dictionary problem. In the dictionary problem, a data structure should maintain a collection of key–value pairs subject to operations that insert or delete pairs from the collection or that search for the value associated with a given key. In open addressing solutions to this problem, the data structure is an array  $T$  (the hash table) whose cells  $T$ . (when nonempty) each store a single key–value pair. A hash function is used to map each key into the cell of  $T$  where that key should be stored, typically scrambling the keys so that keys with similar values are not placed near each other in the table. A hash collision occurs when the hash function maps a key into a cell that is already occupied by a different key. Linear probing is a strategy for resolving collisions, by placing the new key into the closest following empty cell.

**Code :**



# Implement the following for Hashing:

# Write a program to implement the concept of linear probing.

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, 32, 34, 67, 77, 45, 54]
    list_of_list_index = [None] * len(list_of_keys)
    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))
```



**Output :**

```
===== RESTART: C:\Users\user\abhay\python\DataStructures\Practical7b
Before : [None, None, None, None, None, None, 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 :7
hash value for 32 is :2
hash value for 34 is :4
Collision detected for 34
hash value for 67 is :7
Collision detected for 67
hash value for 77 is :7
Collision detected for 77
hash value for 45 is :5
Collision detected for 45
hash value for 54 is :4
Collision detected for 54
After: [54, 1, 32, 23, 43, 34, 45, 87, 67, 77]
>>> |
```

## **Practical No. 8**

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

**Github Repository Link :**

<https://github.com/AJ-Walker/DS/blob/master/Practical8.py>

### **Theory :**

A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child. The topmost node is called root of the tree. The elements that are directly under an element are called its children. The element directly above something is called its parent. Finally, elements with no children are called leaves.

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 non-decreasing 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 on of an expression tree.

#### Uses of Postorder

Postorder traversal is used to delete the tree. Please see [the question for deletion of tree](#) for details. Postorder traversal is also useful to get the postfix expression of an expression tree.

### **Code :**

\*Practical8.py - C:\Users\user\abhay\python\DataStructures\Practical8.py (3.8.1)\*

File Edit Format Run Options Window Help

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

class Node:

def \_\_init\_\_(self,data):

self.data = data

self.left = None

self.right = None

def insert(self,data):

if self.data:

if data < self.data:

if self.left is None:

self.left = Node(data)

else:

self.left.insert(data)

elif data > self.data:

if self.right is None:

self.right = Node(data)

else:

self.right.insert(data)

else:

self.data = data

def preorder(self):

print("preorder : ", self.data)

if self.left:

self.left.preorder()

if self.right:

self.right.preorder()

def inorder(self):

if self.left:

self.left.inorder()

print("inorder : ",self.data)

if self.right:

self.right.inorder()

def postorder(self):

if self.left:

self.left.postorder()

if self.right:

self.right.postorder()

print("postorder : ",self.data)

def PrintTree(self):

print(self.data)

if self.left:

self.left.PrintTree()

if self.right:

self.right.PrintTree()

tree = Node(25)

tree.insert(23)

tree.insert(45)

tree.insert(29)

tree.insert(46)

tree.PrintTree()

tree.preorder()

tree.inorder()

tree.postorder()

## Output :

===== RESTART: C:\Users\user\abhay\python\DataStructur

25

23

45

29

46

preorder : 25

preorder : 23

preorder : 45

preorder : 29

preorder : 46

inorder : 23

inorder : 25

inorder : 29

inorder : 45

inorder : 46

postorder : 23

postorder : 29

postorder : 46

postorder : 45

postorder : 25

~~~~~