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. Devansh Solani

Roll No: 360 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 CS during the academic year 2020-21.

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

External Examiner

Mr. Gangashankar Singh
(Subject-In-Charge)

Date of Examination: (College Stamp)

Class: S.Y. B.Sc. CS Sem- III **Roll No: _360______**

Subject: Data Structures

INDEX

Sr No	Date	Topic	Sign
1	04/09/2020	Implement the following for Array: a) Write a program to store the elements in 1-D array and provide an option to perform the operations like searching, sorting, merging, reversing the elements. b) Write a program to perform the Matrix addition, Multiplication and Transpose Operation.	
2	11/09/2020	Implement Linked List. Include options for insertion, deletion and search of a number, reverse the list and concatenate two linked lists.	
3	18/09/2020	Implement the following for Stack: a) Perform Stack operations using Array implementation. b. b) Implement Tower of Hanoi. c) WAP to scan a polynomial using linked list and add two polynomials. d) WAP to calculate factorial and to compute the factors of a given no. (i) using recursion, (ii) using iteration	
4	25/09/2020	Perform Queues operations using Circular Array implementation.	
5	01/10/2020	Write a program to search an element from a list. Give user the option to perform Linear or Binary search.	
6	09/10/2020	WAP to sort a list of elements. Give user the option to perform sorting using Insertion sort, Bubble sort or Selection sort.	
7	16/10/2020	Implement the following for Hashing: a) Write a program to implement the collision technique. b) Write a program to implement the concept of linear probing.	
8	23/10/2020	Write a program for inorder, postorder and preorder traversal of tree.	

Practical 1

Aim: Implement the following for Array:

Theory:

An array is a special variable, which can hold more than one value at a time. An array can hold many values under a single name, and you can access the values by referring to an index number. You can use the for in loop to loop through all the elements of an array. You can use the append() method to add an element to an array. You can use the pop() method to remove an element from the array. You can also use the remove() method to remove an element from the array.

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.

Code:

```
class OneD:
    def __init__(self, a):
        self.array = a

    def search(self, e):
        if e in self.array:
            return True
        return False

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

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

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

a = [5,6,7,89,2,5,6,1]
o = OneD(a)
print(o.sort())
print(o.search(89))
print(o.merg([8,5,7,9,3]))
print(o.reverse())
```

Output:

```
[1, 2, 5, 5, 6, 6, 7, 89]
True
[1, 2, 5, 5, 6, 6, 7, 89, 8, 5, 7, 9, 3]
[3, 9, 7, 5, 8, 89, 7, 6, 6, 5, 5, 2, 1]
```

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

Code:

```
import numpy as np
M1 = [[8, 14, -6],
           [12,7,4],
           [-11,3,21]]
M2 = [[3, 16, -6],
           [9,7,-4],
           [-1,3,13]]
M3  = [[0,0,0],
            [0,0,0],
            [0,0,0]]
matrix_length = len(M1)

#To Add M1 and M2 matrices
for i in range(len(M1)):
    for k in range(len(M2)):
            M3[i][k] = M1[i][k] + M2[i][k]
print("The sum of Matrix M1 and M2 = ", M3)

#To Multiply M1 and M2 matrices
for i in range(len(M1)):
    for k in range(len(M2)):
            M3[i][k] = M1[i][k] * M2[i][k]
print("The multiplication of Matrix M1 and M2 = ", M3)

#To Matrix Transpose
M1 = np.array([[3, 6, 9], [5, -10, 15], [4,8,12]])
M2 = M1.transpose()

print("Transpose of Matrix M1 is: ", M2)
```

Output:

```
The sum of Matrix M1 and M2 =  [[11, 30, -12], [21, 14, 0], [-12, 6, 34]]
The multiplication of Matrix M1 and M2 =  [[24, 224, 36], [108, 49, -16], [11, 9, 273]]
Transpose of Matrix M1 is:  [[  3   5   4]
 [  6 -10   8]
 [  9  15  12]]
```

Practical 2

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

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 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. A linked list is created by using the node class we studied in the last chapter. We create a Node object and create another class to use this ode object. Singly linked lists can be traversed in only forward direction starting form 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. Inserting an element in the linked list involves reassigning the pointers from the existing nodes to the newly inserted node. You can remove an existing node using the key for that node.

Code:

```
class Node:
    def __init__(self, data=None):
        self.data = data
        self.next = None

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

    def Atbegining(self, data_in):
        NewNode = Node(data_in)
        NewNode.next = self.head
        self.head = NewNode
# Function to remove node
    def RemoveNode(self, Removekey):
        HeadVal = self.head
        if (HeadVal is not None):
            if (HeadVal.data == Removekey):
                self.head = HeadVal.next
                HeadVal = None
                return
        while (HeadVal is not None):
            if HeadVal.data == Removekey:
                break
            prev = HeadVal
            HeadVal = HeadVal.next
        if (HeadVal == None):
            return
        prev.next = HeadVal.next
        HeadVal = None
    def LListprint(self):
        printval = self.head
        while (printval):
            print(printval.data),
            printval = printval.next

llist = SLinkedList()
llist.Atbegining("Mon")
llist.Atbegining("Tue")
llist.Atbegining("Wed")
llist.Atbegining("Thu")
llist.RemoveNode("Tue")
llist.LListprint()
```

Output:

```
Thu
Wed
Mon
```

Practical 3

Aim: Implement the following for Stack:

Theory:

In English dictionary the word stack means arranging objects on over another. It is the same way memory is allocated in this data structure. It stores the data elements in a similar fashion as a bunch of plates are stored one above another in the kitchen. So, stack data structure allows operations at one end which can be called top of the stack. We can add elements or remove elements only form this end of the stack. In a stack the element inserted last in sequence will come out first as we can remove only from the top of the stack. Such feature is known as Last in First Out(LIFO) feature. The operations of adding and removing the elements is known as PUSH and POP. In the following program we implement it as add and remove functions. We declare an empty list and use the append() and pop() methods to add and remove the data elements. The remove function in the following program returns the top most element.

A) Aim: Perform Stack operations using Array implementation.

Code and Output:

In [16]:
```
class Stack:
     def __init__(self):
         self.items = []

     def isEmpty(self):
         return self.items == []

     def push(self, item):
         self.items.append(item)

     def pop(self):
         return self.items.pop()

     def size(self):
         return len(self.items)

s=Stack()
print(s.isEmpty())
s.push(4)
s.push(8)
s.push(10)
s.push('hey')
s.push('hola')
print(s.size())
print(s.isEmpty())
s.push(8.4)
print(s.pop())
print(s.pop())
print(s.size())
```

```
True
5
False
8.4
hola
4
```

C) Aim: WAP to scan a polynomial using linked list and add two polynomials.

Code and Output:

```
In [17]: def addPolynom(A, B, m, n):
                 size = max(m, n);
                 sum = [0 for i in range(size)]
                 for i in range(0, m, 1):
                         sum[i] = A[i]
                         for i in range(n):
                                 sum[i] += B[i]
                                 return sum
         def dispPoly(poly, n):
                 for i in range(n):
                         print(poly[i], end = "")
                         if (i != 0):
                                 print("x^", i, end = "")
                                 if (i != n - 1):
                                         print(" + ", end = "")
         if __name__ == '__main__':
                 A = [5, 0, 10, 6]
                 B = [1, 2, 4]
                 m = len(A)
                 n = len(B)
                 print("First polynomial is: ")
                 dispPoly(A, m)
                 print("\n", end = "")
                 print("Second polynomial is: ")
                 dispPoly(B, n)
                 print("\n", end = "")
                 sum = addPolynom(A, B, m, n)
                 size = max(m, n)
                 print("Sum of polynomial is: ")
                 dispPoly(sum, size)

         First polynomial is:
         50x^ 1 + 10x^ 2 + 6x^ 3
         Second polynomial is:
         12x^ 1 + 4x^ 2
         Sum of polynomial is:
         60x^ 1 + 0x^ 2 + 0x^ 3
```

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

Code and Output:

```
In [8]: def recur_factorial(num):
            if num == 1:
                return num
            else:
                return num*recur_factorial(num-1)
```

```
In [10]: def iter_factorial(num):
             fact = 1
             while num >= 1:
                 # multiply current number with the product of previous numbers
                 fact = fact * num
                 # reduce the number by 1
                 num = num - 1
             return fact
```

```
In [11]: num = int(input("Enter a number: "))
         if num < 0:
             print("Sorry, factorial does not exist for negative numbers")
         elif num == 0:
             print("The factorial of 0 is 1")
         else:
             print("The factorial of",num,"using recursion is",recur_factorial(num))
             print("The factorial of",num,"using iteration is",iter_factorial(num))

         Enter a number: 5
         The factorial of 5 using recursion is 120
         The factorial of 5 using iteration is 120
```

```
In [12]: def print_factors(num):
             print("The factors of",num,"are:")
             for i in range(1, num + 1):
                 if num % i == 0:
                     print(i)

         print_factors(num)

         The factors of 5 are:
         1
         5
```

Practical 4

Aim: Perform Queues operations using Circular Array implementation.

Theory:

An array is called circular if we consider first element as next of last element. Circular arrays are used to implement queue. The approach takes of O(n) time but takes extra space of order O(n). An efficient solution is to deal with circular arrays using the same array. If a careful observation is run through the array, then after n^{th} index, the next index always starts from 0 so using mod operator, we can easily access the elements of the circular list, if we use (i)%n and run the loop from i^{th} index to $n+i^{th}$ index. and apply mod we can do the traversal in a circular array within the given array without using any extra space.

Code and Output:

```
In [4]: # Circular Queue implementation in Python

class MyCircularQueue():

    def __init__(self, k):
        self.k = k
        self.queue = [None] * k
        self.head = self.tail = -1

    # Insert an element into the circular queue
    def enqueue(self, data):

        if ((self.tail + 1) % self.k == self.head):
            print("The circular queue is full\n")

        elif (self.head == -1):
            self.head = 0
            self.tail = 0
            self.queue[self.tail] = data
        else:
            self.tail = (self.tail + 1) % self.k
            self.queue[self.tail] = data

    # Delete an element from the circular queue
    def dequeue(self):
        if (self.head == -1):
            print("The circular queue is empty\n")

        elif (self.head == self.tail):
            temp = self.queue[self.head]
            self.head = -1
            self.tail = -1
            return temp
        else:
            temp = self.queue[self.head]
            self.head = (self.head + 1) % self.k
            return temp

    def printCQueue(self):
        if(self.head == -1):
            print("No element in the circular queue")

        elif (self.tail >= self.head):
            for i in range(self.head, self.tail + 1):
                print(self.queue[i], end=" ")
            print()
        else:
            for i in range(self.head, self.k):
                print(self.queue[i], end=" ")
            for i in range(0, self.tail + 1):
                print(self.queue[i], end=" ")
            print()


obj = MyCircularQueue(5)
obj.enqueue(1)
obj.enqueue(2)
obj.enqueue(3)
obj.enqueue(4)
obj.enqueue(5)
print("Initial queue")
obj.printCQueue()

obj.dequeue()
print("After removing an element from the queue")
obj.printCQueue()
```

```
Initial queue
1 2 3 4 5
After removing an element from the queue
2 3 4 5
```

Practical 5

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

Theory:

An array is called circular if we consider first element as next of last element. Circular arrays are used to implement queue. The approach takes of O(n) time but takes extra space of order O(n). An efficient solution is to deal with circular arrays using the same array. If a careful observation is run through the array, then after n^{th} index, the next index always starts from 0 so using mod operator, we can easily access the elements of the circular list, if we use (i)%n and run the loop from i^{th} index to $(n+i)^{th}$ index . and apply mod we can do the traversal in a circular array within the given array without using any extra space**.**

Code and Output:

In [21]:
```
class search:
    def __init__(self, l, e, type):
        self.l = l
        self.e = e
        self.type = type
        if type == 'l' or 'L':
            if self.linear():
                print('Element Present at ', self.linear())
            else:
                print("Element Not there!")
        elif type == 'B' or 'b':
            if self.binary() != -1:
                print("Element is present at index", str(self.binary()))
            else:
                print("Element is not present in array")
        else:
            print('Enter a valide type of Search')

    def linear(self):
        for i in range(len(self.l)):
            if self.l[i] == self.e:
                return i
        return False

    def binary(self):
        low = 0
        high = len(self.l) - 1
        mid = 0

        while low <= high:
            mid = (high + low) // 2
            if self.l[mid] < self.e:
                low = mid + 1
            elif self.l[mid] > self.e:
                high = mid - 1
            else:
                return mid
        return -1


a = [1,8,3,4,5,9,2,7]
print(search(a, 5,'b'))
```

```
Element Present at  4
```

Practical 6

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

Theory:

Selection sort is to repetitively pick up the smallest element and put it into the right position. A loop through the array finds the smallest element easily. After the smallest element is put in the first position, it is fixed and then we can deal with the rest of the array. The following implementation uses a nested loop to repetitively pick up the smallest element and swap it to its final position. The swap() method exchanges two elements in an array Output:. Insertion sort maintains a sorted sub-array, and repetitively inserts new elements into it. Suppose the array length is n. The outer loop runs roughly n times, and the inner loop on average runs n/2 times. The total time is about t(n) = n * (n/2) = O(n^2). In terms of the efficiency, this is the same as selection sort. Bubble sort repetitively compares adjacent pairs of elements and swaps if necessary. Suppose the array length is n. The outer loop runs roughly n times, and the inner loop on average runs n/2 times. The total time is about t(n) = n * (n/2) = O(n^2). In terms of the efficiency, this is the same as selection sort and insertion sort.

Code and Output:

```
In [23]: class sort:
    def __init__(self,arr,type):
        self.arr = arr
        self.type = type
        if self.type == 'i' or 'I':
            self.insertionSort()
        elif self.type == 'b' or 'B':
            self.bubbleSort()
        elif self.type == 's' or 'S':
            self.selection()
        else:
            print('Invalid Sort type')

    def insertionSort(self):
        print('Using Insertion Sort')
        for i in range(1, len(self.arr)):
            key = self.arr[i]
            j = i-1
            while j >=0 and key < self.arr[j] :
                    self.arr[j+1] = self.arr[j]
                    j -= 1
            self.arr[j+1] = key

    def bubbleSort(self):
        print('Using Bubble Sort')
        n = len(self.arr)
        for i in range(n-1):
            for j in range(0, n-i-1):
                if self.arr[j] > self.arr[j+1] :
                    self.arr[j], self.arr[j+1] = self.arr[j+1], self.arr[j]

    def selection(self):
        print('Using Selection Sort')
        for i in range(len(self.arr)):
            min_ = i
            for j in range(i+1, len(self.arr)):
                if self.arr[min_] > self.arr[j]:
                    min_ = j
            self.arr[i], self.arr[min_] = self.arr[min_], self.arr[i]

a = [7,8,6,2,3,5,7,2,0,1,3]
sort(a, 'I')
print(a)
```

```
Using Insertion Sort
[0, 1, 2, 2, 3, 3, 5, 6, 7, 7, 8]
```

Practical 7

Aim: Implement the following for Hashing:

Theory:

Hash Table: An array that stores pointers to records corresponding to a given phone number. An entry in hash table is NIL if no existing phone number has hash function value equal to the index for the entry. Collision Handling: Since a hash function gets us a small number for a big key, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique. Following are the ways to handle collisions: Chaining: The idea is to make each cell of hash table point to a linked list of records that have same hash function value. Chaining is simple, but requires additional memory outside the table. Open Addressing: In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we one by one examine table slots until the desired element is found or it is clear that the element is not in the table.

A) Aim: Write a program to implement the collision technique.

Code and Output:

```
In [8]: def display_hash(hashTable):

            for i in range(len(hashTable)):
                print(i, end = " ")

                for j in hashTable[i]:
                    print("-->", end = " ")
                    print(j, end = " ")

                print()
        HashTable = [[] for _ in range(10)]
        def Hashing(keyvalue):
            return keyvalue % len(HashTable)
        def insert(Hashtable, keyvalue, value):

            hash_key = Hashing(keyvalue)
            Hashtable[hash_key].append(value)

        insert(HashTable, 10, 'Allahabad')
        insert(HashTable, 25, 'Mumbai')
        insert(HashTable, 20, 'Mathura')
        insert(HashTable, 9, 'Delhi')
        insert(HashTable, 21, 'Punjab')
        insert(HashTable, 21, 'Noida')

        display_hash (HashTable)
```

```
0 --> Allahabad --> Mathura
1 --> Punjab --> Noida
2
3
4
5 --> Mumbai
6
7
8
9 --> Delhi
```

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

Code and Output:

```
list_ = [113 , 117 , 97 , 100 , 114 , 108 , 116 , 105 , 99]


hash_values = []
def hash_func(list_):
    list_2 = [None for i in range(11)]
    for i in list_:
        #print(i % len(list_2))
        hash_values.append(i % len(list_2))
        list_2[i % len(list_2)] = i
    print(list_2)
    print(list_)
    print(hash_values)
    print(116 % 11)
    print(97 % 11)



print(hash_func(list_))
```

```
[99, 100, None, 113, 114, None, 105, 117, None, 108, None]
[113, 117, 97, 100, 114, 108, 116, 105, 99]
[3, 7, 9, 1, 4, 9, 6, 6, 0]
6
9
None
```

Practical 8

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

Theory:

Tree represents the nodes connected by edges. It is a non-linear data structure. It has the following properties. 1. One node is marked as Root node. 2. Every node other than the root is associated with one parent node. 3. Each node can have an arbitrary number of child node. We create a tree data structure in python by using the concept of nodes. We designate one node as root node and then add more nodes as child nodes. To insert into a tree we use the same node class created above and add an insert method to it The insert method compares the value of the node to the parent node and decides to add it as a left node or a right node. Finally, the PrintTree method is used to print the tree. A tree can be traversed by deciding on a sequence to visit each node. As we can clearly see we can start at a node then visit the left sub-tree first and right sub-tree next. Or we can also visit the right sub-tree first and left sub-tree next. Accordingly, there are different names for these tree traversal methods.

Code and Output:

In [2]:
```
class Queue(object):
    def __init__(self):
        self.items = []

    def enqueue(self, item):
        self.items.insert(0, item)

    def dequeue(self):
        if not self.is_empty():
            return self.items.pop()

    def is_empty(self):
        return len(self.items) == 0

    def peek(self):
        if not self.is_empty():
            return self.items[-1].value

    def __len__(self):
        return self.size()

    def size(self):
        return len(self.items)


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

```python
class BinaryTree(object):
    def __init__(self, root):
        self.root = Node(root)

    def print_tree(self, traversal_type):
        if traversal_type == "preorder":
            return self.preorder_print(tree.root, "")
        elif traversal_type == "inorder":
            return self.inorder_print(tree.root, "")
        elif traversal_type == "postorder":
            return self.postorder_print(tree.root, "")
#         elif traversal_type == "levelorder":
#             return self.levelorder_print(tree.root)

        else:
            print("Traversal type " + str(traversal_type) + " is not supported.")
            return False

    def preorder_print(self, start, traversal):
        """Root->Left->Right"""
        if start:
            traversal += (str(start.value) + "-")
            traversal = self.preorder_print(start.left, traversal)
            traversal = self.preorder_print(start.right, traversal)
        return traversal

    def inorder_print(self, start, traversal):
        """Left->Root->Right"""
        if start:
            traversal = self.inorder_print(start.left, traversal)
            traversal += (str(start.value) + "-")
            traversal = self.inorder_print(start.right, traversal)
        return traversal

    def postorder_print(self, start, traversal):
        """Left->Right->Root"""
        if start:
            traversal = self.inorder_print(start.left, traversal)
            traversal = self.inorder_print(start.right, traversal)
            traversal += (str(start.value) + "-")
        return traversal
```

```python
tree = BinaryTree(1)
tree.root.left = Node(2)
tree.root.right = Node(3)
tree.root.left.left = Node(4)
tree.root.left.right = Node(5)

print(tree.print_tree("preorder"))
print(tree.print_tree("inorder"))
print(tree.print_tree("postorder"))
```

```
1-2-4-5-3-
4-2-5-1-3-
4-2-5-3-1-
```

Practical 9

Aim: Write a program to generate the adjacency matrix.

Theory:

In graph theory and computer science, an adjacency matrix is a square matrix used to represent a finite graph. The elements of the matrix indicate whether pairs of vertices are adjacent or not in the graph. In the special case of a finite simple graph, the adjacency matrix is a (0,1)-matrix with zeros on its diagonal. If the graph is undirected (i.e. all of its edges are bidirectional), the adjacency matrix is symmetric. The relationship between a graph and the eigenvalues and eigenvectors of its adjacency matrix is studied in spectral graph theory. The adjacency matrix of a graph should be distinguished from its incidence matrix, a different matrix representation whose elements indicate whether vertex–edge pairs are incident or not, and its degree matrix, which contains information about the degree of each vertex.

Code and Output:

In [27]:
```
class Graph(object):

    # Initialize the matrix
    def __init__(self, size):
        self.adjMatrix = []
        for i in range(size):
            self.adjMatrix.append([0 for i in range(size)])
        self.size = size

    # Add edges
    def add_edge(self, v1, v2):
        if v1 == v2:
            print("Same vertex %d and %d" % (v1, v2))
        self.adjMatrix[v1][v2] = 1
        self.adjMatrix[v2][v1] = 1

    # Remove edges
    def remove_edge(self, v1, v2):
        if self.adjMatrix[v1][v2] == 0:
            print("No edge between %d and %d" % (v1, v2))
            return
        self.adjMatrix[v1][v2] = 0
        self.adjMatrix[v2][v1] = 0

    def __len__(self):
        return self.size

    # Print the matrix
    def print_matrix(self):
        for row in self.adjMatrix:
            for val in row:
                print('{:4}'.format(val)),
            print


def main():
    g = Graph(5)
    g.add_edge(0, 1)
    g.add_edge(0, 2)
    g.add_edge(1, 2)
    g.add_edge(2, 0)
    g.add_edge(2, 3)

    g.print_matrix()
```

```
if __name__ == '__main__':
    main()
```

```
0
1
1
0
0
1
0
1
0
0
1
1
0
1
0
0
0
1
0
0
0
0
0
0
0
```

Practical 10

Aim: Write a program for shortest path diagram.

Theory:

In graphs, to reach from one point to another, we use shortest path algorithms. There are many algorithms to do it. Few of them are Breath First Search, Depth First Search, Djiktras Algorithm, etx. Djikstra's algorithm is a path-finding algorithm, like those used in routing and navigation. We will be using it to find the shortest path between two nodes in a graph. It fans away from the starting node by visiting the next node of the lowest weight and continues to do so until the next node of the lowest weight is the end node.

Code and Output:

```
In [33]: def BFS_SP(graph, start, goal):
    explored = []
    queue = [[start]]
    if start == goal:
        print("Same Node")
        return
    while queue:
        path = queue.pop(0)
        node = path[-1]
        if node not in explored:
            neighbours = graph[node]
            for neighbour in neighbours:
                new_path = list(path)
                new_path.append(neighbour)
                queue.append(new_path)
                if neighbour == goal:
                    print("Shortest path = ", *new_path)
                    return
            explored.append(node)
    print("So sorry, but a connecting path doesnt exist :")
    return

if __name__ == "__main__":
    graph = {'A': ['B', 'E', 'C'],
             'B': ['A', 'D', 'E'],
             'C': ['A', 'F', 'G'],
             'D': ['B', 'E'],
             'E': ['A', 'B', 'D'],
             'F': ['C'],
             'G': ['C']}
    BFS_SP(graph, 'A', 'D')

Shortest path =  A B D
```