```python
from Tree import Node
import Stack
import Stack
import Arrays
from Arrays import Array
import sys
import time
import SinglyLinkedList
from collections import defaultdict
import sys

**SORTING ALGORITHMS**

**BUBBLE SORT**

def bubbleSort(arr):
    n = len(arr)
    for i in range(n):
        swapped = False
        for j in range(0, n-i-1):
            if arr[j] > arr[j+1]:
                arr[j], arr[j+1] = arr[j+1], arr[j]
                swapped = True
        if swapped == False:
            break


arr = [64, 34, 25, 12, 22, 11, 90]
bubbleSort(arr)
print("Sorted array:")
for x in arr:
    print(x, end=' ')
'''
Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

Worst and Average Case Time Complexity: O(n*n). Worst case occurs when array is reverse sorted.

Best Case Time Complexity: O(n). Best case occurs when array is already sorted.

Auxiliary Space: O(1)

Boundary Cases: Bubble sort takes minimum time (Order of n) when elements are already sorted.

Sorting In Place: Yes

Stable: Yes

Due to its simplicity, bubble sort is often used to introduce the concept of a sorting algorithm.
```

```
In computer graphics it is popular for its capability to detect a very small error (like swap of just two elements) in almost-sorted arrays and fix it with just linear complexity (2n). For example, it is used in a polygon filling algorithm, where bounding lines are sorted by their x coordinate at a specific scan line (a line parallel to x axis) and with incrementing y their order changes (two elements are swapped) only at intersections of two lines
'''

**SELECTION SORT**


def selectionSort(arr):
    n = len(arr)
    for i in range(n):
        min_idx = i
        for j in range(i+1, n):
            if arr[min_idx] > arr[j]:
                min_idx = j
        arr[i], arr[min_idx] = arr[min_idx], arr[i]


arr = [64, 25, 12, 22, 11]
selectionSort(arr)
for x in arr:
    print(x, end=' ')

'''
The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning. The algorithm maintains two subarrays in a given array.

Time Complexity: O(n2) as there are two nested loops.

Auxiliary Space: O(1)
The good thing about selection sort is it never makes more than O(n) swaps and can be useful when memory write is a costly operation.

Stability : The default implementation is not stable. However it can be made stable. Please see stable selection sort for details.

In Place : Yes, it does not require extra space
'''

**INSERTION SORT**


def insertionSort(arr):
    n = len(arr)
    for i in range(1, n):
        key = arr[i]
        j = i-1
        while j >= 0 and key < arr[j]:
            arr[j+1] = arr[j]
            j -= 1
        arr[j+1] = key

```

```

arr = [12, 11, 13, 5, 6]
insertionSort(arr)
for x in arr:
    print(x, end=' ')
'''
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.

Time Complexity: O(n*2)

Auxiliary Space: O(1)

Boundary Cases: Insertion sort takes maximum time to sort if elements are sorted in reverse order. And it takes minimum time (Order of n) when elements are already sorted.

Sorting In Place: Yes

Stable: Yes

Uses: Insertion sort is used when number of elements is small. It can also be useful when input array is almost sorted, only few elements are misplaced in complete big array.
'''

**MERGE SORT**


def mergeSort(arr):
    n = len(arr)
    if n > 1:
        mid = len(arr)//2
        L = arr[:mid]
        R = arr[mid:]
        mergeSort(L)
        mergeSort(R)
        i = j = k = 0

        while i < len(L) and j < len(R):
            if L[i] < R[j]:
                arr[k] = L[i]
                i += 1
            else:
                arr[k] = R[j]
                j += 1
            k += 1

        while i < len(L):
            arr[k] = L[i]
            i += 1
            k += 1

        while j < len(R):
            arr[k] = R[j]
            j += 1
```

```python
            k += 1


arr = [12, 11, 13, 5, 6, 7]
mergeSort(arr)
for x in arr:
    print(x, end=' ')
'''
Merge Sort is a Divide and Conquer algorithm. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves. The merge() function is used for merging two halves. The merge(arr, l, m, r) is a key process that assumes that arr[l..m] and arr[m+1..r] are sorted and merges the two sorted sub-arrays into one.

Time Complexity: Sorting arrays on different machines. Merge Sort is a recursive algorithm and time complexity can be expressed as following recurrence relation.
T(n) = 2T(n/2) + θ(n)

The above recurrence can be solved either using the Recurrence Tree method or the Master method. It falls in case II of Master Method and the solution of the recurrence is θ(nLogn). Time complexity of Merge Sort is θ(nLogn) in all 3 cases (worst, average and best) as merge sort always divides the array into two halves and takes linear time to merge two halves.
Auxiliary Space: O(n)
Algorithmic Paradigm: Divide and Conquer
Sorting In Place: No in a typical implementation
Stable: Yes

Applications of Merge Sort

   - Merge Sort is useful for sorting linked lists in O(nLogn) time.In the case of linked lists, the case is different mainly due to the difference in memory allocation of arrays and linked lists. Unlike arrays, linked list nodes may not be adjacent in memory. Unlike an array, in the linked list, we can insert items in the middle in O(1) extra space and O(1) time. Therefore, the merge operation of merge sort can be implemented without extra space for linked lists.
   - In arrays, we can do random access as elements are contiguous in memory. Let us say we have an integer (4-byte) array A and let the address of A[0] be x then to access A[i], we can directly access the memory at (x + i*4). Unlike arrays, we can not do random access in the linked list. Quick Sort requires a lot of this kind of access. In a linked list to access i’th index, we have to travel each and every node from the head to i’th node as we don’t have a continuous block of memory. Therefore, the overhead increases for quicksort. Merge sort accesses data sequentially and the need of random access is low.
   - Inversion Count Problem
   - Used in External Sorting

Drawbacks of Merge Sort

   - Slower comparative to the other sort algorithms for smaller tasks.
    Merge sort algorithm requires additional memory space of 0(n) for the temporary array .
   - It goes through the whole process even if the  array is sorted.

```

```python
'''

**QUICK SORT**


def partition(arr, low, high):
    i = (low-1)
    pivot = arr[high]
    for j in range(low, high):
        if arr[j] < pivot:
            i = i+1
            arr[i], arr[j] = arr[j], arr[i]
    arr[i+1], arr[high] = arr[high], arr[i+1]
    return (i+1)


def quickSort(arr, low, high):
    if low < high:
        pi = partition(arr, low, high)
        quickSort(arr, low, pi-1)
        quickSort(arr, pi+1, high)


arr = [10, 7, 8, 9, 1, 5]
n = len(arr)
quickSort(arr, 0, n-1)
for x in arr:
    print(x, end=' ')

'''
QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

    Always pick first element as pivot.
    Always pick last element as pivot (implemented below)
    Pick a random element as pivot.
    Pick median as pivot.

The key process in quickSort is partition(). Target of partitions is, given an array and an element x of array as pivot, put x at its correct position in sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.
Solution of above recurrence is also O(nLogn)
Although the worst case time complexity of QuickSort is O(n2) which is more than many other sorting algorithms like Merge Sort and Heap Sort, QuickSort is faster in practice, because its inner loop can be efficiently implemented on most architectures, and in most real-world data. QuickSort can be implemented in different ways by changing the choice of pivot, so that the worst case rarely occurs for a given type of data. However, merge sort is generally considered better when data is huge and stored in external storage.

Is QuickSort stable?
The default implementation is not stable. However any sorting algorithm can be made stable by considering indexes as comparison parameter.

```

```
Is QuickSort In-place?
As per the broad definition of in-place algorithm it qualifies as an in-place sorting algorithm as it uses extra space only for storing recursive function calls but not for manipulating the input.
'''

**HEAP SORT**


def heapify(arr, n, i):
    largest = i
    l = 2*i+1
    r = 2*i+2
    if l < n and arr[largest] < arr[l]:
        largest = l
    if r < n and arr[largest] < arr[r]:
        largest = r
    if largest != i:
        arr[i], arr[largest] = arr[largest], arr[i]
        heapify(arr, n, largest)


def heapSort(arr):
    n = len(arr)
    for i in range(n//2 - 1, -1, -1):
        heapify(arr, n, i)
    for i in range(n-1, 0, -1):
        arr[i], arr[0] = arr[0], arr[i]
        heapify(arr, i, 0)


arr = [12, 68, 8, 9, 1, 5]
heapSort(arr)
for x in arr:
    print(x, end=' ')

'''
Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for the remaining elements.

Heap sort is an in-place algorithm.
Its typical implementation is not stable, but can be made stable (See this)

Time Complexity: Time complexity of heapify is O(Logn). Time complexity of createAndBuildHeap() is O(n) and overall time complexity of Heap Sort is O(nLogn).

Applications of HeapSort
1. Sort a nearly sorted (or K sorted) array
2. k largest(or smallest) elements in an array
Heap sort algorithm has limited uses because Quicksort and Mergesort are better in practice. Nevertheless, the Heap data structure itself is enormously used.
'''
```

```python

class Graph:
    def __init__(self, vertices):
        self.graph = defaultdict(list)
        self.V = vertices

    def addEdge(self, u, v):
        self.graph[u].append(v)

    def topologicalSortUtil(self, v, visited, stack):
        visited[v] = True
        for i in self.graph[v]:
            if visited[i] == False:
                self.topologicalSortUtil(i, visited, stack)
        stack.append(v)

    def topologicalSort(self):
        visited = [False]*self.V
        stack = []
        for i in range(self.V):
            if visited[i] == False:
                self.topologicalSortUtil(i, visited, stack)
        print(stack[::-1])


g = Graph(6)
g.addEdge(5, 2)
g.addEdge(5, 0)
g.addEdge(4, 0)
g.addEdge(4, 1)
g.addEdge(2, 3)
g.addEdge(3, 1)
print("Topological Sort for this graph:")

'''
Topological sorting for Directed Acyclic Graph (DAG) is a linear ordering of vertices such that for every directed edge u v, vertex u comes before v in the ordering. Topological Sorting for a graph is not possible if the graph is not a DAG.

For example, a topological sorting of the following graph is “5 4 2 3 1 0”. There can be more than one topological sorting for a graph. For example, another topological sorting of the following graph is “4 5 2 3 1 0”. The first vertex in topological sorting is always a vertex with in-degree as 0 (a vertex with no incoming edges).

Topological Sorting vs Depth First Traversal (DFS):

In DFS, we print a vertex and then recursively call DFS for its adjacent vertices. In topological sorting, we need to print a vertex before its adjacent vertices. For example, in the given graph, the vertex ‘5’ should be printed before vertex ‘0’, but unlike DFS, the vertex ‘4’ should also be printed before vertex ‘0’. So Topological sorting is different from DFS. For example, a DFS of the shown graph is “5 2 3 1 0 4”, but it is not a topological sorting.

Complexity Analysis:

```

```
    Time Complexity: O(V+E).
    The above algorithm is simply DFS with an extra stack. So time complexity is the same as DFS which is.
    Auxiliary space: O(V).
    The extra space is needed for the stack.

Note: Here, we can also use vector instead of the stack. If the vector is used then print the elements in reverse order to get the topological sorting.

Applications:
Topological Sorting is mainly used for scheduling jobs from the given dependencies among jobs. In computer science, applications of this type arise in instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in make files, data serialization, and resolving symbol dependencies in linkers.
'''


**ARRAY**


class Array(object):
    def __init__(self, sizeOfArray, arrayType=int):
        self.sizeOfArray = len(list(map(arrayType, range(sizeOfArray))))
        self.arrayItems = [arrayType(0)] * sizeOfArray
        self.arrayType = arrayType

    def __str__(self):
        return ' '.join([str(i) for i in self.arrayItems])

    def __len__(self):
        return len(self.arrayItems)

    def __setitem__(self, index, data):
        self.arrayItems[index] = data

    def __getitem__(self, index):
        return self.arrayItems[index]

    def search(self, keyToSearch):
        for i in range(self.sizeOfArray):
            if (self.arrayItems[i] == keyToSearch):
                return i
        return -1

    def insert(self, keyToInsert, position):
        if(self.sizeOfArray > position):
            for i in range(self.sizeOfArray - 2, position - 1, -1):
                self.arrayItems[i + 1] = self.arrayItems[i]
            self.arrayItems[position] = keyToInsert
        else:
            print('Array size is:', self.sizeOfArray)

    def delete(self, keyToDelete, position):
        if(self.sizeOfArray > position):
            for i in range(position, self.sizeOfArray - 1):
```

```
                self.arrayItems[i] = self.arrayItems[i + 1]
            self.arrayItems[i + 1] = self.arrayType(0)
        else:
            print('Array size is:', self.sizeOfArray)


if __name__ == '__main__':
    a = Array(10, int)
    a.insert(2, 2)
    a.insert(3, 1)
    a.insert(4, 7)
    print(len(a))


**REVERSING ARRAY**


def reversingAnArray(start, end, myArray):
    while(start < end):
        myArray[start], myArray[end - 1] = myArray[end - 1], myArray[start]
        start += 1
        end -= 1


if __name__ == '__main__':
    myArray = Arrays.Array(10)
    myArray.insert(2, 2)
    myArray.insert(1, 3)
    myArray.insert(3, 1)
    print('Array before Reversing:', myArray)
    reversingAnArray(0, len(myArray), myArray)
    print('Array after Reversing:', myArray)

**ARRAY ROTATION**


def rotation(rotateBy, myArray):
    for i in range(0, rotateBy):
        rotateOne(myArray)
    return myArray


def rotateOne(myArray):
    for i in range(len(myArray) - 1):
        myArray[i], myArray[i + 1] = myArray[i + 1], myArray[i]


if __name__ == '__main__':
    myArray = Array(10)
    for i in range(len(myArray)):
        myArray.insert(i, i)
    print('Before Rotation:', myArray)
    print('After Rotation:', rotation(3, myArray))

**GET MISSING NUMBER**
```

```
def findMissing(myArray, n):
    n = n - 1
    totalSum = (n * (n + 1)) // 2
    for i in range(0, n):
        totalSum -= myArray[i]
    return totalSum


if __name__ == '__main__':
    myArray = Array(10)
    for i in range(len(myArray)):
        myArray.insert(i, i)
    myArray.delete(4, 4)
    print('Original Array:', myArray)
    print('Missing Element:', findMissing(myArray, len(myArray)))


**ODD NUMBER OF TIMES**


def printOddOccurences(array):
    odd = 0
    for element in array:
        odd = odd ^ element
    return odd


if __name__ == '__main__':
    myArray = [3, 4, 1, 2, 4, 1, 2, 5, 6, 4, 6, 5, 3]
    print(printOddOccurences(myArray))


**CHECK FOR PAIR SUM**


def checkSum(array, sum):
    array = sorted(array)
    leftIndex = 0
    rightIndex = len(array) - 1
    while leftIndex < rightIndex:
        if (array[leftIndex] + array[rightIndex] == sum):
            return array[leftIndex], array[rightIndex]
        elif(array[leftIndex] + array[rightIndex] < sum):
            leftIndex += 1
        else:
            rightIndex += 1
    return False, False


if __name__ == '__main__':
    myArray = [10, 20, 30, 40, 50]
    sum = 80
    number1, number2 = checkSum(myArray, sum)
    if(number1 and number2):
        print('Array has elements:', number1, 'and', number2, 'with sum:', sum)
    else:
        print('Array doesn\'t have elements with the sum:', sum)
```

```

**LONGEST DECREASING SUBSEQUENCE**


def lds(arr, n):
    lds = [0] * n
    max = 0
    for i in range(n):
        lds[i] = 1
    for i in range(1, n):
        for j in range(i):
            if (arr[i] < arr[j] and
                    lds[i] < lds[j] + 1):
                lds[i] = lds[j] + 1
    for i in range(n):
        if (max < lds[i]):
            max = lds[i]
    return max


**MINCOIN**


def min_coins(coins, sum):
    dp = [0 for i in range(sum + 1)]
    dp[0] = 0
    for i in range(1, sum + 1):
        dp[i] = sys.maxsize
    for i in range(1, sum + 1):
        for j in range(len(coins)):
            if (coins[j] <= i):
                res = dp[i - coins[j]]
                if (res != sys.maxsize and res + 1 < dp[i]):
                    dp[i] = res + 1
    return dp[sum]


if __name__ == "__main__":
    coins = [9, 6, 5, 1]
    m = len(coins)
    amount = 11
    print("Minimum coins:", min_coins(coins, amount))

**FIBONACCI**


def fibonacci(number):
    if myList[number] == None:
        myList[number] = fibonacci(number - 1) + fibonacci(number - 2)
    return myList[number]


def fibonacciRec(number):
    if number == 1 or number == 0:
        return number
    else:
        return (fibonacciRec(number - 1) + fibonacciRec(number - 2))

```

```python
def fib_memoization(n, lookup):
    if n == 0 or n == 1:
        lookup[n] = n
    if lookup[n] is None:
        lookup[n] = fib(n-1, lookup) + fib(n-2, lookup)
    return lookup[n]


if __name__ == '__main__':
    userInput = int(input('Enter the number: '))

    myList = [None for _ in range(userInput + 1)]

    myList[0] = 0
    myList[1] = 1

    startTime = time.time()
    result = fibonacci(userInput)
    stopTime = time.time()
    print('Time:', (stopTime - startTime), 'Result:', result)

    startTime = time.time()
    result = fibonacciRec(userInput)
    stopTime = time.time()
    print('Time:', (stopTime - startTime), 'Result:', result)

    startTime = time.time()
    lookup = [None]*(101)
    result = fib_memoization(userInput, lookup)
    stopTime = time.time()
    print('Time:', (stopTime - startTime), 'Result:', result)

**Longest Increasing Subsequence**


def longest_increaing_subsequence(myList):
    lis = [1] * len(myList)
    elements = [0] * len(myList)
    for i in range(1, len(myList)):
        for j in range(0, i):
            if myList[i] > myList[j] and lis[i] < lis[j] + 1:
                lis[i] = lis[j]+1
                elements[i] = j
    idx = 0

    maximum = max(lis)
    idx = lis.index(maximum)
    seq = [myList[idx]]
    while idx != elements[idx]:
        idx = elements[idx]
        seq.append(myList[idx])
    return (maximum, reversed(seq))


myList = [10, 22, 9, 33, 21, 50, 41, 60]
ans = longest_increaing_subsequence(myList)
print('Length of lis is', ans[0])
```

```
print('The longest sequence is', ', '.join(str(x) for x in ans[1]))


**LONGEST CONTINUOUS ODD SUBSEQUENCE**


def longest_continuous_odd_subsequence(array):
    final_list = []
    temp_list = []
    for i in array:
        if i % 2 == 0:
            if temp_list != []:
                final_list.append(temp_list)
            temp_list = []
        else:
            temp_list.append(i)

    if temp_list != []:
        final_list.append(temp_list)
    result = max(final_list, key=len)
    print(result, len(result))


if __name__ == '__main__':
    array = [2, 6, 8, 3, 9, 1, 5, 6, 1, 3, 5, 7, 7, 1, 2, 3, 4, 5]
    longest_continuous_odd_subsequence(array)

**SIEVE OF ERATOSTHENES**


def sieve_of_eratosthenes(n):

    prime = [True for i in range(n+1)]
    p = 2
    while (p * p <= n):
        if (prime[p] == True):
            for i in range(p * 2, n+1, p):
                prime[i] = False
        p += 1
    for p in range(2, n):
        if prime[p]:
            print(p),


if __name__ == '__main__':
    n = 30
    print("Following are the prime numbers smaller"),
    print("than or equal to", n)
    sieve_of_eratosthenes(n)

**GRAPH**


class AdjacencyList(object):
    def __init__(self):
        self.List = {}

    def addEdge(self, fromVertex, toVertex):
```

```

        if fromVertex in self.List.keys():
            self.List[fromVertex].append(toVertex)
        else:
            self.List[fromVertex] = [toVertex]

    def printList(self):
        for i in self.List:
            print(i, '->', ' -> '.join([str(j) for j in self.List[i]]))


if __name__ == '__main__':
    al = AdjacencyList()
    al.addEdge(0, 1)
    al.addEdge(0, 4)
    al.addEdge(4, 1)
    al.addEdge(4, 3)
    al.addEdge(1, 0)
    al.addEdge(1, 4)
    al.addEdge(1, 3)
    al.addEdge(1, 2)
    al.addEdge(2, 3)
    al.addEdge(3, 4)
    al.printList()


**DEPTH FIRST SEARCH**


class Graph():
    def __init__(self):
        self.vertex = {}

    def printGraph(self):
        print(self.vertex)
        for i in self.vertex.keys():
            print(i, ' -> ', ' -> '.join([str(j) for j in self.vertex[i]]))

    def addEdge(self, fromVertex, toVertex):
        if fromVertex in self.vertex.keys():
            self.vertex[fromVertex].append(toVertex)
        else:
            self.vertex[fromVertex] = [toVertex]

    def DFS(self):
        visited = [False] * len(self.vertex)
        for i in range(len(self.vertex)):
            if visited[i] == False:
                self.DFSRec(i, visited)

    def DFSRec(self, startVertex, visited):
        visited[startVertex] = True
        print(startVertex, end=' ')
        for i in self.vertex.keys():
            if visited[i] == False:
                self.DFSRec(i, visited)

```

```

if __name__ == '__main__':
    g = Graph()
    g.addEdge(0, 1)
    g.addEdge(0, 2)
    g.addEdge(1, 2)
    g.addEdge(2, 0)
    g.addEdge(2, 3)
    g.addEdge(3, 3)
    g.printGraph()
    print('DFS:')
    g.DFS()


**BREADTH FIRST SEARCH**


class Graph():
    def __init__(self):
        self.vertex = {}

    def printGraph(self):
        for i in self.vertex.keys():
            print(i, ' -> ', ' -> '.join([str(j) for j in self.vertex[i]]))

    def addEdge(self, fromVertex, toVertex):
        if fromVertex in self.vertex.keys():
            self.vertex[fromVertex].append(toVertex)
        else:
            self.vertex[fromVertex] = [toVertex]

    def BFS(self, startVertex):
        visited = [False] * len(self.vertex)
        queue = []
        visited[startVertex] = True
        queue.append(startVertex)
        while queue:
            startVertex = queue.pop(0)
            print(startVertex, end=' ')
            for i in self.vertex[startVertex]:
                if visited[i] == False:
                    queue.append(i)
                    visited[i] = True


if __name__ == '__main__':
    g = Graph()
    g.addEdge(0, 1)
    g.addEdge(0, 2)
    g.addEdge(1, 2)
    g.addEdge(2, 0)
    g.addEdge(2, 3)
    g.addEdge(3, 3)
    g.printGraph()
    print('BFS:')
    g.BFS(2)

```

```
**DETECT CYCLE IN DIRECTED GRAPH**


class Graph():
    def __init__(self):
        self.vertex = {}

    def printGraph(self):
        for i in self.vertex.keys():
            print(i, ' -> ', ' -> '.join([str(j) for j in self.vertex[i]]))

    def addEdge(self, fromVertex, toVertex):
        if fromVertex in self.vertex.keys():
            self.vertex[fromVertex].append(toVertex)
        else:
            self.vertex[fromVertex] = [toVertex]

    def checkCyclic(self):
        visited = [False] * len(self.vertex)
        stack = [False] * len(self.vertex)
        for vertex in range(len(self.vertex)):
            if visited[vertex] == False:
                if self.checkCyclicRec(visited, stack, vertex) == True:
                    return True
        return False

    def checkCyclicRec(self, visited, stack, vertex):
        visited[vertex] = True
        stack[vertex] = True
        for adjacentNode in self.vertex[vertex]:
            if visited[adjacentNode] == False:
                if self.checkCyclicRec(visited, stack, adjacentNode) == True:
                    return True
            elif stack[adjacentNode] == True:
                return True
        stack[vertex] = False
        return False


if __name__ == '__main__':
    graph = Graph()
    graph.addEdge(0, 1)
    graph.addEdge(0, 2)
    graph.addEdge(1, 2)
    graph.addEdge(2, 0)
    graph.addEdge(2, 3)
    graph.addEdge(3, 3)
    graph.printGraph()

    if graph.checkCyclic() == 1:
        print("Graph has a cycle")
    else:
        print("Graph has no cycle")
```

```
**DETECT CYCLE IN UNDIRECTED GRAPH**


class Graph():
    def __init__(self):
        self.vertex = {}

    def printGraph(self):
        for i in self.vertex.keys():
            print(i, ' -> ', ' -> '.join([str(j) for j in self.vertex[i]]))

    def addEdge(self, fromVertex, toVertex):
        if fromVertex in self.vertex.keys() and toVertex in self.vertex.keys():
            self.vertex[fromVertex].append(toVertex)
            self.vertex[toVertex].append(fromVertex)
        else:
            self.vertex[fromVertex] = [toVertex]
            self.vertex[toVertex] = [fromVertex]

    def checkCyclic(self):
        visited = [False] * len(self.vertex)
        for vertex in range(len(self.vertex)):
            if visited[vertex] == False:
                if self.checkCyclicRec(visited, -1, vertex) == True:
                    return True
        return False

    def checkCyclicRec(self, visited, parent, vertex):
        visited[vertex] = True
        for adjacentNode in self.vertex[vertex]:
            if visited[adjacentNode] == False:
                if self.checkCyclicRec(visited, vertex, adjacentNode) == True:
                    return True
            elif parent != adjacentNode:
                return True
        return False


if __name__ == '__main__':
    graph = Graph()
    graph.addEdge(0, 1)
    graph.addEdge(0, 2)
    graph.addEdge(1, 2)
    graph.addEdge(2, 0)
    graph.addEdge(2, 3)
    graph.addEdge(3, 3)
    graph.printGraph()

    if graph.checkCyclic() == 1:
        print("Graph has a cycle")
    else:
        print("Graph has no cycle")

    g1 = Graph()
    g1.addEdge(0, 1)
```

```
    g1.addEdge(1, 2)
    g1.printGraph()

    if g1.checkCyclic() == 1:
        print("Graph has a cycle")
    else:
        print("Graph has no cycle")


**TOPOLOGICAL SORT**


class Graph():
    def __init__(self, count):
        self.vertex = {}
        self.count = count

    def printGraph(self):
        for i in self.vertex.keys():
            print(i, ' -> ', ' -> '.join([str(j) for j in self.vertex[i]]))

    def addEdge(self, fromVertex, toVertex):
        if fromVertex in self.vertex.keys():
            self.vertex[fromVertex].append(toVertex)
        else:
            self.vertex[fromVertex] = [toVertex]
            self.vertex[toVertex] = []

    def topologicalSort(self):
        visited = [False] * self.count
        stack = []
        for vertex in range(self.count):
            if visited[vertex] == False:
                self.topologicalSortRec(vertex, visited, stack)
        print(' '.join([str(i) for i in stack]))

    def topologicalSortRec(self, vertex, visited, stack):
        visited[vertex] = True
        try:
            for adjacentNode in self.vertex[vertex]:
                if visited[adjacentNode] == False:
                    self.topologicalSortRec(adjacentNode, visited, stack)
        except KeyError:
            return
        stack.insert(0, vertex)


if __name__ == '__main__':
    g = Graph(6)
    g.addEdge(5, 2)
    g.addEdge(5, 0)
    g.addEdge(4, 0)
    g.addEdge(4, 1)
    g.addEdge(2, 3)
    g.addEdge(3, 1)
    g.topologicalSort()
```

```

**PRIM'S ALGORITHM**


class Graph(object):
    def __init__(self, vertices):
        self.vertices = vertices
        self.graph = [[0 for column in range(vertices)]
                      for row in range(vertices)]

    def getMinimumKey(self, weight, visited):
        min = 9999
        for i in range(self.vertices):
            if weight[i] < min and visited[i] == False:
                min = weight[i]
                minIndex = i
        return minIndex

    def primsAlgo(self):
        weight = [9999] * self.vertices
        MST = [None] * self.vertices
        weight[0] = 0
        visited = [False] * self.vertices
        MST[0] = -1
        for _ in range(self.vertices):
            minIndex = self.getMinimumKey(weight, visited)
            visited[minIndex] = True
            for vertex in range(self.vertices):
                if self.graph[minIndex][vertex] > 0 and visited[vertex] == False and \
                        weight[vertex] > self.graph[minIndex][vertex]:
                    weight[vertex] = self.graph[minIndex][vertex]
                    MST[vertex] = minIndex
        self.printMST(MST)

    def printMST(self, MST):
        print("Edge \tWeight")
        for i in range(1, self.vertices):
            print(MST[i], "-", i, "\t", self.graph[i][MST[i]])


if __name__ == '__main__':
    g = Graph(6)

    g.graph = [[0, 3, 2, 5, 7, 3],
               [3, 0, 4, 8, 6, 6],
               [2, 4, 0, 7, 1, 3],
               [5, 8, 7, 0, 2, 4],
               [7, 6, 1, 2, 0, 3],
               [3, 6, 3, 4, 3, 0]]

    g.primsAlgo()


**KRUSKAL'S ALOGORITHM**


class Graph:
```

```python
    def __init__(self, vertices):
        self.vertices = vertices
        self.graph = []

    def addEdge(self, fromEdge, toEdge, weight):
        self.graph.append([fromEdge, toEdge, weight])

    def find(self, parent, i):
        if parent[i] == i:
            return i
        return self.find(parent, parent[i])

    def union(self, parent, rank, first, second):
        root_x = self.find(parent, first)
        root_y = self.find(parent, second)

        if rank[root_x] < rank[root_y]:
            parent[root_x] = root_y
        elif rank[root_x] > rank[root_y]:
            parent[root_y] = root_x

        elif rank[root_x] == rank[root_y]:
            parent[root_y] = root_x
            rank[root_x] += 1

    def kruskals(self):
        result = []
        sortedIndex = 0
        resultIndex = 0
        self.graph = sorted(self.graph, key=lambda item: item[2])
        parent = []
        rank = []
        for node in range(self.vertices):
            parent.append(node)
            rank.append(0)

        while resultIndex < self.vertices - 1:
            fromEdge, toEdge, weight = self.graph[sortedIndex]
            sortedIndex += 1
            root_x = self.find(parent, fromEdge)
            root_y = self.find(parent, toEdge)
            if root_x != root_y:
                resultIndex += 1
                result.append([fromEdge, toEdge, weight])
                self.union(parent, rank, root_x, root_y)

        print('Constructed Kruskal\'s Minimum Spanning Tree: ')
        for u, v, weight in result:
            print('{} -> {} = {}'.format(u, v, weight))


if __name__ == '__main__':
    g = Graph(4)
    g.addEdge(0, 1, 10)
    g.addEdge(0, 2, 6)
    g.addEdge(0, 3, 5)
    g.addEdge(1, 3, 15)
    g.addEdge(2, 3, 4)
```

```
    g.kruskals()

**HEAP**


class BinaryHeap(object):
    def __init__(self):
        self.heap = [0]
        self.currentSize = 0

    def __repr__(self):
        heap = self.heap[1:]
        return ' '.join(str(i) for i in heap)

    def shiftUp(self, index):
        while (index // 2) > 0:
            if self.heap[index] < self.heap[index // 2]:
                temp = self.heap[index // 2]
                self.heap[index // 2] = self.heap[index]
                self.heap[index] = temp
            index = index // 2

    def insert(self, key):
        self.heap.append(key)
        self.currentSize += 1
        self.shiftUp(self.currentSize)

    def shiftDown(self, index):
        while(index * 2) <= self.currentSize:
            minimumChild = self.minChild(index)
            if self.heap[index] > self.heap[minimumChild]:
                temp = self.heap[index]
                self.heap[index] = self.heap[minimumChild]
                self.heap[minimumChild] = temp
            index = minimumChild

    def minChild(self, i):
        if i * 2 + 1 > self.currentSize:
            return i * 2
        else:
            if self.heap[i * 2] < self.heap[i * 2 + 1]:
                return i * 2
            else:
                return i * 2 + 1

    def delete(self):
        deletedNode = self.heap[1]
        self.heap[1] = self.heap[self.currentSize]
        self.currentSize -= 1
        self.heap.pop()
        self.shiftDown(1)
        return deletedNode

    def buildHeap(self, alist):
        i = len(alist) // 2
        self.currentSize = len(alist)
        self.heap = [0] + alist[:]
        while (i > 0):
```

```
            self.shiftDown(i)
            i = i - 1


bh = BinaryHeap()
bh.buildHeap([9, 5, 6, 2, 3])

print('Deleted:', bh.delete())
print('Deleted:', bh.delete())
print('Deleted:', bh.delete())
bh.insert(3)
print('Deleted:', bh.delete())
print(bh)


**HEAP SORT**


def HeapSort(alist):
    heapify(alist)
    end = len(alist) - 1
    while end > 0:
        alist[end], alist[0] = alist[0], alist[end]
        shiftDown(alist, 0, end - 1)
        end -= 1


def heapify(alist):
    ''' This function helps to maintain the heap property '''
    start = len(alist) // 2
    while start >= 0:
        shiftDown(alist, start, len(alist) - 1)
        start -= 1


def shiftDown(alist, start, end):
    root = start
    while root * 2 + 1 <= end:
        child = root * 2 + 1

        if child + 1 <= end and alist[child] < alist[child + 1]:
            child += 1

        if child <= end and alist[root] < alist[child]:
            alist[root], alist[child] = alist[child], alist[root]
            root = child
        else:
            return


if __name__ == '__main__':
    alist = [12, 2, 4, 5, 2, 3]
    HeapSort(alist)
    print('Sorted Array:', alist)

**MAX HEAP**
```

```python
def heapify(A):
    '''Turns a list `A` into a max-ordered binary heap.'''
    n = len(A) - 1
    for node in range(n/2, -1, -1):
        __shiftdown(A, node)
    return


def push_heap(A, val):
    '''Pushes a value onto the heap `A` while keeping the heap property
    intact.  The heap size increases by 1.'''
    A.append(val)
    __shiftup(A, len(A) - 1)
    return


def pop_heap(A):
    '''Returns the max value from the heap `A` while keeping the heap
    property intact.  The heap size decreases by 1.'''
    n = len(A) - 1
    __swap(A, 0, n)
    max = A.pop(n)
    __shiftdown(A, 0)
    return max


def replace_key(A, node, newval):
    '''Replace the key at node `node` in the max-heap `A` by `newval`.
    The heap size does not change.'''
    curval = A[node]
    A[node] = newval

    if newval > curval:
        __shiftup(A, node)

    elif newval < curval:
        __shiftdown(A, node)
    return


def __swap(A, i, j):
    A[i], A[j] = A[j], A[i]
    return


def __shiftdown(A, node):
    '''Traverse down a binary tree `A` starting at node `node` and
    turn it into a max-heap'''
    child = 2*node + 1

    if child > len(A) - 1:
        return

    if (child + 1 <= len(A) - 1) and (A[child+1] > A[child]):
        child += 1

    if A[node] < A[child]:
        __swap(A, node, child)
```

```python
        __shiftdown(A, child)
    else:
        return


def __shiftup(A, node):
    '''Traverse up an otherwise max-heap `A` starting at node `node`
    (which is the only node that breaks the heap property) and restore
    the heap structure.'''
    parent = (node - 1)/2
    if A[parent] < A[node]:
        __swap(A, node, parent)

    if parent <= 0:
        return
    else:
        __shiftup(A, parent)


**SINGLY LINKED LIST**


class Node(object):

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

    def setData(self, data):
        self.data = data

    def getData(self):
        return self.data

    def setNext(self, next):
        self.next = next

    def getNext(self):
        return self.next


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

    def printLinkedList(self):
        temp = self.head
        while(temp):
            print(temp.data, end=' ')
            temp = temp.next

    def insertAtStart(self, data):
        if self.head == None:
            newNode = Node(data)
            self.head = newNode
        else:
            newNode = Node(data)
            newNode.next = self.head
```

```python
            self.head = newNode

    def insertBetween(self, previousNode, data):
        if (previousNode.next is None):
            print('Previous node should have next node!')
        else:
            newNode = Node(data)
            newNode.next = previousNode.next
            previousNode.next = newNode

    def insertAtEnd(self, data):
        newNode = Node(data)
        temp = self.head
        while(temp.next != None):
            temp = temp.next
        temp.next = newNode

    def delete(self, data):
        temp = self.head
        if (temp.next is not None):
            if(temp.data == data):
                self.head = temp.next
                temp = None
                return
            else:
                while(temp.next != None):
                    if(temp.data == data):
                        break
                    prev = temp
                    temp = temp.next
                if temp == None:
                    return
                prev.next = temp.next
                return

    def search(self, node, data):
        if node == None:
            return False
        if node.data == data:
            return True
        return self.search(node.getNext(), data)


if __name__ == '__main__':
    List = LinkedList()
    List.head = Node(1)
    node2 = Node(2)
    List.head.setNext(node2)
    node3 = Node(3)
    node2.setNext(node3)
    List.insertAtStart(4)
    List.insertBetween(node2, 5)
    List.insertAtEnd(6)
    List.printLinkedList()
    print()
    List.delete(3)
    List.printLinkedList()
    print()
```

```
    print(List.search(List.head, 1))

**CIRCULAR LINKED LIST**


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


class CreateList:
    def __init__(self):
        self.head = Node(None)
        self.tail = Node(None)
        self.head.next = self.tail
        self.tail.next = self.head

    def add(self, data):
        newNode = Node(data)
        if self.head.data is None:
            self.head = newNode
            self.tail = newNode
            newNode.next = self.head
        else:
            self.tail.next = newNode
            self.tail = newNode
            self.tail.next = self.head

    def display(self):
        current = self.head
        if self.head is None:
            print("List is empty")
            return
        else:
            print("Nodes of the circular linked list: ")
            print(current.data),
            while(current.next != self.head):
                current = current.next
                print(current.data),


class CircularLinkedList:
    cl = CreateList()
    cl.add(1)
    cl.add(2)
    cl.add(3)
    cl.add(4)
    cl.display()


**DOUBLY LINKED LIST**


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

```python
        self.previous = previous


class DoublyLinkedList(object):
    def __init__(self):
        self.head = None

    def insertAtStart(self, data):
        if self.head == None:
            newNode = Node(data)
            self.head = newNode
        else:
            newNode = Node(data)
            self.head.previous = newNode
            newNode.next = self.head
            self.head = newNode

    def insertAtEnd(self, data):
        newNode = Node(data)
        temp = self.head
        while(temp.next != None):
            temp = temp.next
        temp.next = newNode
        newNode.previous = temp

    def delete(self, data):
        temp = self.head
        if(temp.next != None):

            if(temp.data == data):
                temp.next.previous = None
                self.head = temp.next
                temp.next = None
                return
            else:
                while(temp.next != None):
                    if(temp.data == data):
                        break
                    temp = temp.next
                if(temp.next):

                    temp.previous.next = temp.next
                    temp.next.previous = temp.previous
                    temp.next = None
                    temp.previous = None
                else:

                    temp.previous.next = None
                    temp.previous = None
                return

        if (temp == None):
            return

    def printdll(self):
        temp = self.head
        while(temp != None):
            print(temp.data, end=' ')
```

```
            temp = temp.next


if __name__ == '__main__':
    dll = DoublyLinkedList()
    dll.insertAtStart(1)
    dll.insertAtStart(2)
    dll.insertAtEnd(3)
    dll.insertAtStart(4)
    dll.printdll()
    dll.delete(2)
    print()
    dll.printdll()

**LENGTH OF LINKED LIST**


def checkLength(linkedList):
    count = 0
    temp = linkedList.head
    while(temp != None):
        count += 1
        temp = temp.next
    return count


if __name__ == '__main__':
    myLinkedList = SinglyLinkedList.LinkedList()
    for i in range(10):
        myLinkedList.insertAtStart(i)
    myLinkedList.printLinkedList()
    print()
    print('Length:', checkLength(myLinkedList))


**REVERSING LINKED LIST**


def reverseLinkedList(myLinkedList):
    previous = None
    current = myLinkedList.head
    while(current != None):
        temp = current.next
        current.next = previous
        previous = current
        current = temp
    myLinkedList.head = previous


if __name__ == '__main__':
    myLinkedList = SinglyLinkedList.LinkedList()
    for i in range(10, 0, -1):
        myLinkedList.insertAtStart(i)
    print('Original:', end=' ')
    myLinkedList.printLinkedList()
    print()
    print('Reversed:', end=' ')
    reverseLinkedList(myLinkedList)
```

```
    myLinkedList.printLinkedList()


**QUEUE**


class Queue(object):
    def __init__(self, limit=10):
        self.queue = []
        self.front = None
        self.rear = None
        self.limit = limit
        self.size = 0

    def __str__(self):
        return ' '.join([str(i) for i in self.queue])

    def isEmpty(self):
        return self.size <= 0

    def enqueue(self, data):
        if self.size >= self.limit:
            return -1
        else:
            self.queue.append(data)
        if self.front is None:
            self.front = self.rear = 0
        else:
            self.rear = self.size
        self.size += 1

    def dequeue(self):
        if self.isEmpty():
            return -1
        else:
            self.queue.pop(0)
            self.size -= 1
            if self.size == 0:
                self.front = self.rear = 0
            else:
                self.rear = self.size - 1

    def getSize(self):
        return self.size


if __name__ == '__main__':
    myQueue = Queue()
    for i in range(10):
        myQueue.enqueue(i)
    print(myQueue)
    print('Queue Size:', myQueue.getSize())
    myQueue.dequeue()
    print(myQueue)
    print('Queue Size:', myQueue.getSize())

**DEQUE**

```

```

class Deque(object):
    def __init__(self, limit=10):
        self.queue = []
        self.limit = limit

    def __str__(self):
        return ' '.join([str(i) for i in self.queue])

    def isEmpty(self):
        return len(self.queue) <= 0

    def isFull(self):
        return len(self.queue) >= self.limit

    def insertRear(self, data):
        if self.isFull():
            return
        else:
            self.queue.insert(0, data)

    def insertFront(self, data):
        if self.isFull():
            return
        else:
            self.queue.append(data)

    def deleteRear(self):
        if self.isEmpty():
            return
        else:
            return self.queue.pop(0)

    def deleteFront(self):
        if self.isFull():
            return
        else:
            return self.queue.pop()


if __name__ == '__main__':
    myDeque = Deque()
    myDeque.insertFront(1)
    myDeque.insertRear(2)
    myDeque.insertFront(3)
    myDeque.insertRear(10)
    print(myDeque)
    myDeque.deleteRear()
    print(myDeque)
    myDeque.deleteFront()
    print(myDeque)

**CIRCULAR QUEUE**


class CircularQueue(object):
    def __init__(self, limit=10):
        self.limit = limit
```

```
        self.queue = [None for i in range(limit)]
        self.front = self.rear = -1

    def __str__(self):
        if (self.rear >= self.front):
            return ' '.join([str(self.queue[i]) for i in range(self.front, self.rear + 1)])

        else:
            q1 = ' '.join([str(self.queue[i])
                           for i in range(self.front, self.limit)])
            q2 = ' '.join([str(self.queue[i])
                           for i in range(0, self.rear + 1)])
            return q1 + ' ' + q2

    def isEmpty(self):
        return self.front == -1

    def isFull(self):
        return (self.rear + 1) % self.limit == self.front

    def enqueue(self, data):
        if self.isFull():
            print('Queue is Full!')
        elif self.isEmpty():
            self.front = 0
            self.rear = 0
            self.queue[self.rear] = data
        else:
            self.rear = (self.rear + 1) % self.limit
            self.queue[self.rear] = data

    def dequeue(self):
        if self.isEmpty():
            print('Queue is Empty!')
        elif (self.front == self.rear):
            self.front = -1
            self.rear = -1
        else:
            self.front = (self.front + 1) % self.limit


if __name__ == '__main__':
    myCQ = CircularQueue(5)
    myCQ.enqueue(14)
    myCQ.enqueue(22)
    myCQ.enqueue(13)
    myCQ.enqueue(16)
    print('Queue:', myCQ)
    myCQ.dequeue()
    myCQ.dequeue()
    print('Queue:', myCQ)
    myCQ.enqueue(9)
    myCQ.enqueue(20)
    myCQ.enqueue(5)
    myCQ.enqueue(5)
    print('Queue:', myCQ)

```

```
**PRIORITY QUEUE**


class PriorityQueue(object):
    def __init__(self):
        self.queue = []

    def __str__(self):
        return ' '.join([str(i) for i in self.queue])

    def isEmpty(self):
        return len(self.queue) == []

    def insert(self, data):
        self.queue.append(data)

    def delete(self):
        try:
            max = 0
            for i in range(len(self.queue)):
                if self.queue[i] > self.queue[max]:
                    max = i
            item = self.queue[max]
            del self.queue[max]
            return item
        except IndexError:
            print()
            exit()


if __name__ == '__main__':
    myQueue = PriorityQueue()
    myQueue.insert(12)
    myQueue.insert(1)
    myQueue.insert(14)
    myQueue.insert(7)
    print(myQueue)
    while not myQueue.isEmpty():
        print(myQueue.delete(), end=' ')

**SEGMENT TREE**


class SegmentTree:
    def __init__(self, values):
        self.valarr = values
        self.arr = dict()

    def buildTree(self, start, end, node):
        if start == end:
            self.arr[node] = self.valarr[start]
            return
        mid = (start+end)//2
        self.buildTree(start, mid, node*2)
        self.buildTree(mid+1, end, node*2+1)
        self.arr[node] = self.arr[node*2]+self.arr[node*2+1]

    def rangeQuery(self, node, start, end, l, r):
```

```
        if (l <= start and r >= end):
            return self.arr[node]

        if (end < l or start > r):
            return 0

        mid = (start+end)//2
        return self.rangeQuery(2*node, start, mid, l, r) + self.rangeQuery(2*node+1, mid+1, end, l, r)

    def update(self, node, newvalue, oldvalue, position, start, end):

        if start <= position <= end:
            self.arr[node] += (newvalue-oldvalue)

        if start != end:
            mid = (start+end)//2
            self.update(node*2, newvalue, oldvalue, position, start, mid)
            self.update(node*2+1, newvalue, oldvalue, position, mid+1, end)


if __name__ == '__main__':
    l = list(
        map(int, input("Enter the elements of the array separated by space:\n").split()))
    st = SegmentTree(l)
    st.buildTree(0, len(l)-1, 1)
    baseindex = 1
    endindex = len(l)
    print(st.arr)
    print("Sum of numbers from index 3 and 5 is: ",
          st.rangeQuery(1, baseindex, endindex, 3, 5))
    updateindex = 3
    updatevalue = 10
    st.update(1, updatevalue, l[updateindex-1],
              updateindex, baseindex, endindex)

    print("Updated sum of numbers from index 3 and 5 is: ",
          st.rangeQuery(1, baseindex, endindex, 3, 5))

**SEGMENT TREE 2**


class SegmentTree:
    def __init__(self, values):
        self.minarr = dict()

        self.originalarr = values[:]

    def buildminTree(self, start, end, node):
        if start == end:
            self.minarr[node] = self.originalarr[start]
            return
        mid = (start + end) // 2
        self.buildminTree(start, mid, node*2)
        self.buildminTree(mid + 1, end, node*2 + 1)
```

```
        self.minarr[node] = min(self.minarr[node*2], self.minarr[node*2 + 1])

    def minRangeQuery(self, node, start, end, l, r):
        if l <= start and end <= r:
            return self.minarr[node]
        if r < start or l > end:
            return sys.maxsize
        mid = (start+end)//2
        return min(self.minRangeQuery(node*2, start, mid, l, r), self.minRangeQuery(node*2+1, mid+1, end, l, r))

    def update(self, node, newvalue, position, start, end):
        if start <= position <= end:
            self.minarr[node] = min(self.minarr[node], newvalue)

        if start != end:
            mid = (start + end) // 2
            self.update(node * 2, newvalue, position, start, mid)
            self.update(node * 2 + 1, newvalue, position, mid + 1, end)


arr = [10, 5, 9, 3, 4, 8, 6, 7, 2, 1]
st = SegmentTree(arr)
st.buildminTree(0, len(arr)-1, 1)
print("Segment Tree for given array", st.minarr)
print("Minimum of numbers from index 6 to 9 is: ",
      st.minRangeQuery(1, 1, len(arr), 6, 9))
st.update(1, 2, 4, 1, len(arr))
print(st.minarr)
print("Updated minimum of numbers from index 2 to 9 is: ",
      st.minRangeQuery(1, 1, len(arr), 2, 6))


**STACK**


class Stack(object):
    def __init__(self, limit=10):
        self.stack = []
        self.limit = limit

    def __str__(self):
        return ' '.join([str(i) for i in self.stack])

    def push(self, data):
        if len(self.stack) >= self.limit:
            print('Stack Overflow')
        else:
            self.stack.append(data)

    def pop(self):
        if len(self.stack) <= 0:
            return -1
        else:
            return self.stack.pop()

    def peek(self):
```

```
        if len(self.stack) <= 0:
            return -1
        else:
            return self.stack[len(self.stack) - 1]

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

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


if __name__ == '__main__':
    myStack = Stack()
    for i in range(10):
        myStack.push(i)
    print(myStack)
    myStack.pop()
    print(myStack)
    myStack.peek()
    myStack.isEmpty()
    myStack.size()

**INFIX TO POSTFIX**


def isOperand(char):
    return (ord(char) >= ord('a') and ord(char) <= ord('z')) or (ord(char) >= ord('A') and ord(char) <= ord('Z'))


def precedence(char):
    if char == '+' or char == '-':
        return 1
    elif char == '*' or char == '/':
        return 2
    elif char == '^':
        return 3
    else:
        return -1


def infixToPostfix(myExp, myStack):
    postFix = []
    for i in range(len(myExp)):
        if (isOperand(myExp[i])):
            postFix.append(myExp[i])
        elif(myExp[i] == '('):
            myStack.push(myExp[i])
        elif(myExp[i] == ')'):
            topOperator = myStack.pop()
            while(not myStack.isEmpty() and topOperator != '('):
                postFix.append(topOperator)
                topOperator = myStack.pop()
        else:
            while (not myStack.isEmpty()) and (precedence(myExp[i]) <= precedence(myStack.peek())):
                postFix.append(myStack.pop())
```

```
            myStack.push(myExp[i])

    while(not myStack.isEmpty()):
        postFix.append(myStack.pop())
    return ' '.join(postFix)


if __name__ == '__main__':
    myExp = 'a+b*(c^d-e)^(f+g*h)-i'
    myExp = [i for i in myExp]
    print('Infix:', ' '.join(myExp))
    myStack = Stack.Stack(len(myExp))
    print('Postfix:', infixToPostfix(myExp, myStack))


**BALANCED PARANTHESIS**


def parseParenthesis(string):
    balanced = 1
    index = 0
    myStack = Stack.Stack(len(string))
    while (index < len(string)) and (balanced == 1):
        check = string[index]
        if check == '(':
            myStack.push(check)
        else:
            if myStack.isEmpty():
                balanced = 0
            else:
                myStack.pop()
        index += 1

    if balanced == 1 and myStack.isEmpty():
        return True
    else:
        return False


if __name__ == '__main__':
    print(parseParenthesis('((()))'))
    print(parseParenthesis('((())'))

**DECIMAL TO BINARY**


def dtob(decimal, base=2):
    myStack = Stack.Stack()
    while decimal > 0:
        myStack.push(decimal % base)
        decimal //= base

    result = ''
    while not myStack.isEmpty():
        result += str(myStack.pop())

    return result

```

```python

if __name__ == '__main__':
    print(dtob(15))

**REVERSE STRING**


def reverse(string):
    myStack = Stack.Stack(len(string))
    for i in string:
        myStack.push(i)
    result = ''
    while not myStack.isEmpty():
        result += myStack.pop()
    return result


if __name__ == '__main__':
    print(reverse('omkar'))

**QUEUE IMPLEMENTATION USING TWO STACKS**


class StackedQueue:

    def __init__(self):
        self.stack = Stack()
        self.alternateStack = Stack()

    def enqueue(self, item):
        while(not self.stack.is_empty()):
            self.alternateStack.push(self.stack.pop())
        self.alternateStack.push(item)
        while(not self.alternateStack.is_empty()):
            self.stack.push(self.alternateStack.pop())

    def dequeue(self):
        return self.stack.pop()

    def __repr__(self):
        return repr(self.stack)


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

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

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

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

    def is_empty(self):
        return self.items == []
```

```
    def __repr__(self):
        return str(self.items)


if __name__ == "__main__":
    structure = StackedQueue()
    structure.enqueue(4)
    structure.enqueue(3)
    structure.enqueue(2)
    structure.enqueue(1)
    print(structure)
    structure.dequeue()
    print(structure)

**TREE**


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

    def setLeft(self, node):
        self.left = node

    def setRight(self, node):
        self.right = node

    def getLeft(self):
        return self.left

    def getRight(self):
        return self.right

    def setData(self, data):
        self.data = data

    def getData(self):
        return self.data


def inorder(Tree):
    if Tree:
        inorder(Tree.getLeft())
        print(Tree.getData(), end=' ')
        inorder(Tree.getRight())
    return


def preorder(Tree):
    if Tree:
        print(Tree.getData(), end=' ')
        preorder(Tree.getLeft())
        preorder(Tree.getRight())
    return

```

```python

def postorder(Tree):
    if Tree:
        postorder(Tree.getLeft())
        postorder(Tree.getRight())
        print(Tree.getData(), end=' ')
    return


if __name__ == '__main__':
    root = Node(1)
    root.setLeft(Node(2))
    root.setRight(Node(3))
    root.left.setLeft(Node(4))
    print('Inorder  Traversal:')
    inorder(root)
    print('\nPreorder Traversal:')
    preorder(root)
    print('\nPostorder Traversal:')
    postorder(root)


**TREE TRAVERSAL**


class Node(object):
    def __init__(self, val):
        self.key = val
        self.left = None
        self.right = None


class Tree:
    def __init__(self, val):
        self.root = Node(val)

    def insertNode(root, val):
        if(root == None):
            root = Node(val)
        elif(root.key < val):
            root.right = Tree.insertNode(root.right, val)
        else:
            root.left = Tree.insertNode(root.left, val)
        return root

    def inorder(root):
        if(root == None):
            return ""
        else:
            return str(Tree.inorder(root.left)) + " " + str(root.key) + " " + str(Tree.inorder(root.right))

    def preorder(root):
        if(root == None):
            return ""
        else:
            return str(root.key) + " " + str(Tree.preorder(root.left)) + " " + str(Tree.preorder(root.right))
```

```
    def postorder(root):
        if(root == None):
            return ""
        else:
            return str(Tree.postorder(root.left)) + " " + str(Tree.postorder(root.right)) + " " + str(root.key)


array = [1, 22, 3, 44, 32, 35]
treeRoot = Node(array[0])
for i in range(1, len(array)):
    treeRoot = Tree.insertNode(treeRoot, array[i])
print("Inorder:", Tree.inorder(treeRoot))
print("Preorder:", Tree.preorder(treeRoot))
print("Postorder:", Tree.postorder(treeRoot))


**ZIGZAG TRAVERSAL**


class Node:

    def __init__(self, data):
        self.left = None
        self.right = None
        self.data = data


def make_tree() -> Node:
    root = Node(1)
    root.left = Node(2)
    root.right = Node(3)
    root.left.left = Node(4)
    root.left.right = Node(5)
    return root


def zigzag_iterative(root: Node):
    if root == None:
        return
    s1 = []
    s2 = []
    s1.append(root)
    while not len(s1) == 0 or not len(s2) == 0:
        while not len(s1) == 0:
            temp = s1[-1]
            s1.pop()
            print(temp.data, end=" ")
            if temp.left:
                s2.append(temp.left)
            if temp.right:
                s2.append(temp.right)

        while not len(s2) == 0:
            temp = s2[-1]
            s2.pop()
            print(temp.data, end=" ")
```

```python
            if temp.right:
                s1.append(temp.right)
            if temp.left:
                s1.append(temp.left)


def main():
    root = make_tree()
    print("\nZigzag order traversal(iterative) is: ")
    zigzag_iterative(root)
    print()


if __name__ == "__main__":
    import doctest
    doctest.testmod()
    main()


**BINARY SEARCH TREE**


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

    def insert(self, data):
        if self.data == data:
            return False

        elif data < self.data:
            if self.leftChild:
                return self.leftChild.insert(data)
            else:
                self.leftChild = Node(data)
                return True

        else:
            if self.rightChild:
                return self.rightChild.insert(data)
            else:
                self.rightChild = Node(data)
                return True

    def minValueNode(self, node):
        current = node
        while(current.leftChild is not None):
            current = current.leftChild
        return current

    def delete(self, data):
        if self is None:
            return None

        if data < self.data:
            self.leftChild = self.leftChild.delete(data)
```

```
        elif data > self.data:
            self.rightChild = self.rightChild.delete(data)
        else:

            if self.leftChild is None:
                temp = self.rightChild
                self = None
                return temp
            elif self.rightChild is None:
                temp = self.leftChild
                self = None
                return temp

            temp = self.minValueNode(self.rightChild)
            self.data = temp.data
            self.rightChild = self.rightChild.delete(temp.data)
        return self

    def find(self, data):
        if(data == self.data):
            return True
        elif(data < self.data):
            if self.leftChild:
                return self.leftChild.find(data)
            else:
                return False
        else:
            if self.rightChild:
                return self.rightChild.find(data)
            else:
                return False

    def preorder(self):
        if self:
            print(str(self.data), end=' ')
            if self.leftChild:
                self.leftChild.preorder()
            if self.rightChild:
                self.rightChild.preorder()

    def inorder(self):
        if self:
            if self.leftChild:
                self.leftChild.inorder()
            print(str(self.data), end=' ')
            if self.rightChild:
                self.rightChild.inorder()

    def postorder(self):
        if self:
            if self.leftChild:
                self.leftChild.postorder()
            if self.rightChild:
                self.rightChild.postorder()
            print(str(self.data), end=' ')


class Tree(object):
```

```
    def __init__(self):
        self.root = None

    def insert(self, data):
        if self.root:
            return self.root.insert(data)
        else:
            self.root = Node(data)
            return True

    def delete(self, data):
        if self.root is not None:
            return self.root.delete(data)

    def find(self, data):
        if self.root:
            return self.root.find(data)
        else:
            return False

    def preorder(self):
        if self.root is not None:
            print()
            print('Preorder: ')
            self.root.preorder()

    def inorder(self):
        print()
        if self.root is not None:
            print('Inorder: ')
            self.root.inorder()

    def postorder(self):
        print()
        if self.root is not None:
            print('Postorder: ')
            self.root.postorder()


if __name__ == '__main__':
    tree = Tree()
    tree.insert(10)
    tree.insert(12)
    tree.insert(5)
    tree.insert(4)
    tree.insert(20)
    tree.insert(8)
    tree.insert(7)
    tree.insert(15)
    tree.insert(13)
    print(tree.find(1))
    print(tree.find(12))
    '''
    Following tree is getting created:
                    10
                 /      \
               5          12
              / \            \
```

```
            4     8          20
                 /          /
                7         15
                         /
                       13
    '''
    tree.preorder()
    tree.inorder()
    tree.postorder()
    print('\n\nAfter deleting 20')
    tree.delete(20)
    tree.inorder()
    tree.preorder()
    print('\n\nAfter deleting 10')
    tree.delete(10)
    tree.inorder()
    tree.preorder()


**LIST VIEW USING TREE**


class Sample:
    def __init__(self, data_description, node_id, parent_id=""):
        self.data_description = data_description
        self.node_id = node_id
        self.parent_id = parent_id


class Node:
    def __init__(self, data):
        self.data = Sample(data['data_description'],
                           data['node_id'], data['parent_id'])
        self.children = []


class Tree:
    def __init__(self, data):
        self.Root = data

    def insert_child(self, root, new_node):
        if root.data.node_id == new_node.data.parent_id:
            root.children.append(new_node)

        elif len(root.children) > 0:
            for each_child in root.children:
                self.insert_child(each_child, new_node)

    def print_tree(self, root, point):
        print(point, root.data.node_id, root.data.parent_id,
              root.data.data_description)
        if len(root.children) > 0:
            point += "_"
            for each_child in root.children:
                self.print_tree(each_child, point)
```

```
data = {'data_description': 'Sample_root_1', 'node_id': '1', 'parent_id': ''}
data1 = {'data_description': 'Sample_root_2', 'node_id': '2', 'parent_id': '1'}
data2 = {'data_description': 'Sample_root_3', 'node_id': '3', 'parent_id': '1'}
data3 = {'data_description': 'Sample_root_4', 'node_id': '4', 'parent_id': '2'}
data4 = {'data_description': 'Sample_root_5', 'node_id': '5', 'parent_id': '3'}
data5 = {'data_description': 'Sample_root_6', 'node_id': '6', 'parent_id': '4'}
data6 = {'data_description': 'Sample_root_7', 'node_id': '7', 'parent_id': '4'}

a = Tree(Node(data))
a.insert_child(a.Root, Node(data1))
a.insert_child(a.Root, Node(data2))
a.insert_child(a.Root, Node(data3))
a.insert_child(a.Root, Node(data4))
a.insert_child(a.Root, Node(data5))
a.insert_child(a.Root, Node(data6))
a.print_tree(a.Root, "|_")


**BREADTH FIRST TRAVERSAL**


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


def height(node):
    if node is None:
        return 0
    else:
        leftHeight = height(node.leftChild)
        rightHeight = height(node.rightChild)
        if leftHeight > rightHeight:
            return leftHeight + 1
        else:
            return rightHeight + 1


def breadthFirstTraversal(root):
    if root == None:
        return 0
    else:
        h = height(root)
        for i in range(1, h + 1):
            printBFT(root, i)


def printBFT(root, level):
    if root is None:
```

```
            return
        else:
            if level == 1:
                print(root.data, end=' ')
            elif level > 1:
                printBFT(root.leftChild, level - 1)
                printBFT(root.rightChild, level - 1)


if __name__ == '__main__':
    root = Node(1)
    root.leftChild = Node(2)
    root.rightChild = Node(3)
    root.leftChild.leftChild = Node(4)
    breadthFirstTraversal(root)


**COUNT LEAF NODES**


def countLeafNodes(root):
    if root is None:
        return 0
    if(root.left is None and root.right is None):
        return 1
    else:
        return countLeafNodes(root.left) + countLeafNodes(root.right)


if __name__ == '__main__':
    root = Node(1)
    root.setLeft(Node(2))
    root.setRight(Node(3))
    root.left.setLeft(Node(4))
    print('Count of leaf nodes:', countLeafNodes(root))


**TREE FROM INORDER AND PREORDER**


class Node:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None


"""Recursive function to construct binary of size len from
   Inorder traversal in[] and Preorder traversal pre[].  Initial values
   of start and end should be 0 and len -1.  The function doesn't
   do any error checking for cases where inorder and preorder
   do not form a tree """


def buildTree(inOrder, preOrder, start, end):
    if (start > end):
        return None
    tNode = Node(preOrder[buildTree.preIndex])
```

```
    buildTree.preIndex += 1
    if start == end:
        return tNode
    rootIndex = search(inOrder, start, end, tNode.data)
    tNode.left = buildTree(inOrder, preOrder, start, rootIndex-1)
    tNode.right = buildTree(inOrder, preOrder, rootIndex+1, end)
    return tNode


def search(arr, start, end, value):
    for i in range(start, end+1):
        if arr[i] == value:
            return i


def inorder(node):
    if node is None:
        return
    inorder(node.left)
    print(node.data, end=' ')
    inorder(node.right)


inOrder = ['D', 'B', 'E', 'A', 'F', 'C']
preOrder = ['A', 'B', 'D', 'E', 'C', 'F']
buildTree.preIndex = 0
root = buildTree(inOrder, preOrder, 0, len(inOrder)-1)
print("Inorder traversal of the constructed tree is")
inorder(root)


**ROOT TO LEAF PATHS**


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


def printPath(node, path=[]):
    if node is None:
        return
    path.append(node.data)
    if (node.left is None) and (node.right is None):
        print(' '.join([str(i) for i in path if i != 0]))
    else:
        printPath(node.left, path)
        printPath(node.right, path[0:1])


if __name__ == '__main__':
    root = Node(1)
    root.left = Node(2)
    root.right = Node(3)
    root.left.left = Node(4)
    root.right.left = Node(5)
```

```
    printPath(root)


**INORDER PREDECESSOR AND SUCCESSOR**


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

    def findPredecessorAndSuccessor(self, data):
        global predecessor, successor
        predecessor = None
        successor = None
        if self is None:
            return
        if self.data == data:
            if self.left is not None:
                temp = self.left
                if temp.right is not None:
                    while(temp.right):
                        temp = temp.right
                predecessor = temp
            if self.right is not None:
                temp = self.right
                while(temp.left):
                    temp = temp.left
                successor = temp
            return

        if data < self.data:
            print('Left')
            self.left.findPredecessorAndSuccessor(data)
        else:
            print('Right')
            self.right.findPredecessorAndSuccessor(data)

    def insert(self, data):
        if self.data == data:
            return False

        elif data < self.data:
            if self.left:
                return self.left.insert(data)
            else:
                self.left = Node(data)
                return True

        else:
            if self.right:
                return self.right.insert(data)
            else:
                self.right = Node(data)
                return True
```

```
if __name__ == '__main__':
    root = Node(50)
    root.insert(30)
    root.insert(20)
    root.insert(40)
    root.insert(70)
    root.insert(60)
    root.insert(80)
    root.findPredecessorAndSuccessor(70)
    if (predecessor is not None) and (successor is not None):
        print('Predecessor:', predecessor.data, 'Successor:', successor.data)
    else:
        print('Predecessor:', predecessor, 'Successor:', successor)
```