

```

1  from Tree import Node
2  import Stack
3  import Stack
4  import Arrays
5  from Arrays import Array
6  import sys
7  import time
8  import SinglyLinkedList
9  from collections import defaultdict
10 import sys
11
12 **SORTING ALGORITHMS**
13
14 **BUBBLE SORT**
15
16 def bubbleSort(arr):
17     n = len(arr)
18     for i in range(n):
19         swapped = False
20         for j in range(0, n-i-1):
21             if arr[j] > arr[j+1]:
22                 arr[j], arr[j+1] = arr[j+1], arr[j]
23                 swapped = True
24         if swapped == False:
25             break
26
27
28 arr = [64, 34, 25, 12, 22, 11, 90]
29 bubbleSort(arr)
30 print("Sorted array:")
31 for x in arr:
32     print(x, end=' ')
33 '''
34 Bubble Sort is the simplest sorting algorithm that works by repeatedly
35 swapping the adjacent elements if they are in wrong order.
36
37 Worst and Average Case Time Complexity:  $O(n*n)$ . Worst case occurs when
38 array is reverse sorted.
39
40 Best Case Time Complexity:  $O(n)$ . Best case occurs when array is already
41 sorted.
42
43 Auxiliary Space:  $O(1)$ 
44
45 Boundary Cases: Bubble sort takes minimum time (Order of  $n$ ) when elements
46 are already sorted.
47
48 Sorting In Place: Yes
49
50 Stable: Yes
51
52 Due to its simplicity, bubble sort is often used to introduce the concept
53 of a sorting algorithm.

```

```

49 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
50 '''
51
52 **SELECTION SORT**
53
54
55 def selectionSort(arr):
56     n = len(arr)
57     for i in range(n):
58         min_idx = i
59         for j in range(i+1, n):
60             if arr[min_idx] > arr[j]:
61                 min_idx = j
62         arr[i], arr[min_idx] = arr[min_idx], arr[i]
63
64
65 arr = [64, 25, 12, 22, 11]
66 selectionSort(arr)
67 for x in arr:
68     print(x, end=' ')
69
70 '''
71 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.
72
73 Time Complexity:  $O(n^2)$  as there are two nested loops.
74
75 Auxiliary Space:  $O(1)$ 
76 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.
77
78 Stability : The default implementation is not stable. However it can be
    made stable. Please see stable selection sort for details.
79
80 In Place : Yes, it does not require extra space
81 '''
82
83 **INSERTION SORT**
84
85
86 def insertionSort(arr):
87     n = len(arr)
88     for i in range(1, n):
89         key = arr[i]
90         j = i-1
91         while j >= 0 and key < arr[j]:
92             arr[j+1] = arr[j]
93             j -= 1
94         arr[j+1] = key
95

```

```

96
97 arr = [12, 11, 13, 5, 6]
98 insertionSort(arr)
99 for x in arr:
100     print(x, end=' ')
101 '''
102 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.
103
104 Time Complexity: O(n*2)
105
106 Auxiliary Space: O(1)
107
108 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.
109
110 Sorting In Place: Yes
111
112 Stable: Yes
113
114 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.
115 '''
116
117 **MERGE SORT**
118
119
120 def mergeSort(arr):
121     n = len(arr)
122     if n > 1:
123         mid = len(arr)//2
124         L = arr[:mid]
125         R = arr[mid:]
126         mergeSort(L)
127         mergeSort(R)
128         i = j = k = 0
129
130         while i < len(L) and j < len(R):
131             if L[i] < R[j]:
132                 arr[k] = L[i]
133                 i += 1
134             else:
135                 arr[k] = R[j]
136                 j += 1
137             k += 1
138
139         while i < len(L):
140             arr[k] = L[i]
141             i += 1
142             k += 1
143
144         while j < len(R):
145             arr[k] = R[j]
146             j += 1

```

```

147         k += 1
148
149
150 arr = [12, 11, 13, 5, 6, 7]
151 mergeSort(arr)
152 for x in arr:
153     print(x, end=' ')
154 '''
155 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.
156
157 Time Complexity: Sorting arrays on different machines. Merge Sort is a
    recursive algorithm and time complexity can be expressed as following
    recurrence relation.
158  $T(n) = 2T(n/2) + \theta(n)$ 
159
160 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  $\theta(n \log n)$ . Time complexity of Merge Sort is
 $\theta(n \log n)$  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.
161 Auxiliary Space:  $O(n)$ 
162 Algorithmic Paradigm: Divide and Conquer
163 Sorting In Place: No in a typical implementation
164 Stable: Yes
165
166 Applications of Merge Sort
167
168 - Merge Sort is useful for sorting linked lists in  $O(n \log n)$  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.
169 - 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.
170 - Inversion Count Problem
171 - Used in External Sorting
172
173 Drawbacks of Merge Sort
174
175 - Slower comparative to the other sort algorithms for smaller tasks.
176 Merge sort algorithm requires additional memory space of  $O(n)$  for the
    temporary array .
177 - It goes through the whole process even if the array is sorted.
178

```

```

179 '''
180
181 **QUICK SORT**
182
183
184 def partition(arr, low, high):
185     i = (low-1)
186     pivot = arr[high]
187     for j in range(low, high):
188         if arr[j] < pivot:
189             i = i+1
190             arr[i], arr[j] = arr[j], arr[i]
191     arr[i+1], arr[high] = arr[high], arr[i+1]
192     return (i+1)
193
194
195 def quickSort(arr, low, high):
196     if low < high:
197         pi = partition(arr, low, high)
198         quickSort(arr, low, pi-1)
199         quickSort(arr, pi+1, high)
200
201
202 arr = [10, 7, 8, 9, 1, 5]
203 n = len(arr)
204 quickSort(arr, 0, n-1)
205 for x in arr:
206     print(x, end=' ')
207
208 '''
209 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.
210
211     Always pick first element as pivot.
212     Always pick last element as pivot (implemented below)
213     Pick a random element as pivot.
214     Pick median as pivot.
215
216 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.
217 Solution of above recurrence is also  $O(n \log n)$ 
218 Although the worst case time complexity of QuickSort is  $O(n^2)$  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.
219
220 Is QuickSort stable?
221 The default implementation is not stable. However any sorting algorithm
can be made stable by considering indexes as comparison parameter.
222

```

```

223 Is QuickSort In-place?
224 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.
225 '''
226
227 **HEAP SORT**
228
229
230 def heapify(arr, n, i):
231     largest = i
232     l = 2*i+1
233     r = 2*i+2
234     if l < n and arr[largest] < arr[l]:
235         largest = l
236     if r < n and arr[largest] < arr[r]:
237         largest = r
238     if largest != i:
239         arr[i], arr[largest] = arr[largest], arr[i]
240         heapify(arr, n, largest)
241
242
243 def heapSort(arr):
244     n = len(arr)
245     for i in range(n//2 - 1, -1, -1):
246         heapify(arr, n, i)
247     for i in range(n-1, 0, -1):
248         arr[i], arr[0] = arr[0], arr[i]
249         heapify(arr, i, 0)
250
251
252 arr = [12, 68, 8, 9, 1, 5]
253 heapSort(arr)
254 for x in arr:
255     print(x, end=' ')
256
257 '''
258 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.
259
260 Heap sort is an in-place algorithm.
261 Its typical implementation is not stable, but can be made stable (See
    this)
262
263 Time Complexity: Time complexity of heapify is  $O(\log n)$ . Time complexity of
    createAndBuildHeap() is  $O(n)$  and overall time complexity of Heap Sort is
     $O(n \log n)$ .
264
265 Applications of HeapSort
266 1. Sort a nearly sorted (or K sorted) array
267 2. k largest(or smallest) elements in an array
268 Heap sort algorithm has limited uses because Quicksort and Mergesort are
    better in practice. Nevertheless, the Heap data structure itself is
    enormously used.
269 '''
270

```

```

271
272 class Graph:
273     def __init__(self, vertices):
274         self.graph = defaultdict(list)
275         self.v = vertices
276
277     def addEdge(self, u, v):
278         self.graph[u].append(v)
279
280     def topologicalSortUtil(self, v, visited, stack):
281         visited[v] = True
282         for i in self.graph[v]:
283             if visited[i] == False:
284                 self.topologicalSortUtil(i, visited, stack)
285         stack.append(v)
286
287     def topologicalSort(self):
288         visited = [False]*self.v
289         stack = []
290         for i in range(self.v):
291             if visited[i] == False:
292                 self.topologicalSortUtil(i, visited, stack)
293         print(stack[::-1])
294
295
296 g = Graph(6)
297 g.addEdge(5, 2)
298 g.addEdge(5, 0)
299 g.addEdge(4, 0)
300 g.addEdge(4, 1)
301 g.addEdge(2, 3)
302 g.addEdge(3, 1)
303 print("Topological Sort for this graph:")
304
305 '''
306 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.
307
308 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).
309
310 Topological Sorting vs Depth First Traversal (DFS):
311
312 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.
313
314 Complexity Analysis:
315

```

```

316     Time Complexity:  $O(V+E)$ .
317     The above algorithm is simply DFS with an extra stack. So time
complexity is the same as DFS which is.
318     Auxiliary space:  $O(V)$ .
319     The extra space is needed for the stack.
320
321 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.
322
323 Applications:
324 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.
325 '''
326
327
328 **ARRAY**
329
330
331 class Array(object):
332     def __init__(self, sizeofArray, arrayType=int):
333         self.sizeOfArray = len(list(map(arrayType, range(sizeofArray))))
334         self.arrayItems = [arrayType(0)] * sizeofArray
335         self.arrayType = arrayType
336
337     def __str__(self):
338         return ' '.join([str(i) for i in self.arrayItems])
339
340     def __len__(self):
341         return len(self.arrayItems)
342
343     def __setitem__(self, index, data):
344         self.arrayItems[index] = data
345
346     def __getitem__(self, index):
347         return self.arrayItems[index]
348
349     def search(self, keyToSearch):
350         for i in range(self.sizeOfArray):
351             if (self.arrayItems[i] == keyToSearch):
352                 return i
353         return -1
354
355     def insert(self, keyToInsert, position):
356         if(self.sizeOfArray > position):
357             for i in range(self.sizeOfArray - 2, position - 1, -1):
358                 self.arrayItems[i + 1] = self.arrayItems[i]
359                 self.arrayItems[position] = keyToInsert
360         else:
361             print('Array size is:', self.sizeOfArray)
362
363     def delete(self, keyToDelete, position):
364         if(self.sizeOfArray > position):
365             for i in range(position, self.sizeOfArray - 1):

```



```

366         self.arrayItems[i] = self.arrayItems[i + 1]
367         self.arrayItems[i + 1] = self.arrayType(0)
368     else:
369         print('Array size is:', self.sizeOfArray)
370
371
372 if __name__ == '__main__':
373     a = Array(10, int)
374     a.insert(2, 2)
375     a.insert(3, 1)
376     a.insert(4, 7)
377     print(len(a))
378
379
380 **REVERSING ARRAY**
381
382
383 def reversingAnArray(start, end, myArray):
384     while(start < end):
385         myArray[start], myArray[end - 1] = myArray[end - 1],
myArray[start]
386         start += 1
387         end -= 1
388
389
390 if __name__ == '__main__':
391     myArray = Arrays.Array(10)
392     myArray.insert(2, 2)
393     myArray.insert(1, 3)
394     myArray.insert(3, 1)
395     print('Array before Reversing:', myArray)
396     reversingAnArray(0, len(myArray), myArray)
397     print('Array after Reversing:', myArray)
398
399 **ARRAY ROTATION**
400
401
402 def rotation(rotateBy, myArray):
403     for i in range(0, rotateBy):
404         rotateOne(myArray)
405     return myArray
406
407
408 def rotateOne(myArray):
409     for i in range(len(myArray) - 1):
410         myArray[i], myArray[i + 1] = myArray[i + 1], myArray[i]
411
412
413 if __name__ == '__main__':
414     myArray = Array(10)
415     for i in range(len(myArray)):
416         myArray.insert(i, i)
417     print('Before Rotation:', myArray)
418     print('After Rotation:', rotation(3, myArray))
419
420 **GET MISSING NUMBER**
421
422

```

```

423 def findMissing(myArray, n):
424     n = n - 1
425     totalSum = (n * (n + 1)) // 2
426     for i in range(0, n):
427         totalSum -= myArray[i]
428     return totalSum
429
430
431 if __name__ == '__main__':
432     myArray = Array(10)
433     for i in range(len(myArray)):
434         myArray.insert(i, i)
435     myArray.delete(4, 4)
436     print('Original Array:', myArray)
437     print('Missing Element:', findMissing(myArray, len(myArray)))
438
439
440 **ODD NUMBER OF TIMES**
441
442
443 def printOddOccurrences(array):
444     odd = 0
445     for element in array:
446         odd = odd ^ element
447     return odd
448
449
450 if __name__ == '__main__':
451     myArray = [3, 4, 1, 2, 4, 1, 2, 5, 6, 4, 6, 5, 3]
452     print(printOddOccurrences(myArray))
453
454
455 **CHECK FOR PAIR SUM**
456
457
458 def checkSum(array, sum):
459     array = sorted(array)
460     leftIndex = 0
461     rightIndex = len(array) - 1
462     while leftIndex < rightIndex:
463         if (array[leftIndex] + array[rightIndex] == sum):
464             return array[leftIndex], array[rightIndex]
465         elif (array[leftIndex] + array[rightIndex] < sum):
466             leftIndex += 1
467         else:
468             rightIndex -= 1
469     return False, False
470
471
472 if __name__ == '__main__':
473     myArray = [10, 20, 30, 40, 50]
474     sum = 80
475     number1, number2 = checkSum(myArray, sum)
476     if (number1 and number2):
477         print('Array has elements:', number1, 'and', number2, 'with sum:',
sum)
478     else:
479         print('Array doesn\'t have elements with the sum:', sum)

```

```

480
481 **LONGEST DECREASING SUBSEQUENCE**
482
483
484 def lds(arr, n):
485     lds = [0] * n
486     max = 0
487     for i in range(n):
488         lds[i] = 1
489     for i in range(1, n):
490         for j in range(i):
491             if (arr[i] < arr[j] and
492                 lds[i] < lds[j] + 1):
493                 lds[i] = lds[j] + 1
494     for i in range(n):
495         if (max < lds[i]):
496             max = lds[i]
497     return max
498
499
500 **MINCOIN**
501
502
503 def min_coins(coins, sum):
504     dp = [0 for i in range(sum + 1)]
505     dp[0] = 0
506     for i in range(1, sum + 1):
507         dp[i] = sys.maxsize
508     for i in range(1, sum + 1):
509         for j in range(len(coins)):
510             if (coins[j] <= i):
511                 res = dp[i - coins[j]]
512                 if (res != sys.maxsize and res + 1 < dp[i]):
513                     dp[i] = res + 1
514     return dp[sum]
515
516
517 if __name__ == "__main__":
518     coins = [9, 6, 5, 1]
519     m = len(coins)
520     amount = 11
521     print("Minimum coins:", min_coins(coins, amount))
522
523 **FIBONACCI**
524
525
526 def fibonacci(number):
527     if myList[number] == None:
528         myList[number] = fibonacci(number - 1) + fibonacci(number - 2)
529     return myList[number]
530
531
532 def fibonacciRec(number):
533     if number == 1 or number == 0:
534         return number
535     else:
536         return (fibonacciRec(number - 1) + fibonacciRec(number - 2))
537

```

```

538
539 def fib_memoization(n, lookup):
540     if n == 0 or n == 1:
541         lookup[n] = n
542     if lookup[n] is None:
543         lookup[n] = fib(n-1, lookup) + fib(n-2, lookup)
544     return lookup[n]
545
546
547 if __name__ == '__main__':
548     userInput = int(input('Enter the number: '))
549
550     myList = [None for _ in range(userInput + 1)]
551
552     myList[0] = 0
553     myList[1] = 1
554
555     startTime = time.time()
556     result = fibonacci(userInput)
557     stopTime = time.time()
558     print('Time:', (stopTime - startTime), 'Result:', result)
559
560     startTime = time.time()
561     result = fibonacciRec(userInput)
562     stopTime = time.time()
563     print('Time:', (stopTime - startTime), 'Result:', result)
564
565     startTime = time.time()
566     lookup = [None]*(101)
567     result = fib_memoization(userInput, lookup)
568     stopTime = time.time()
569     print('Time:', (stopTime - startTime), 'Result:', result)
570
571 **Longest Increasing Subsequence**
572
573
574 def longest_increaing_subsequence(myList):
575     lis = [1] * len(myList)
576     elements = [0] * len(myList)
577     for i in range(1, len(myList)):
578         for j in range(0, i):
579             if myList[i] > myList[j] and lis[i] < lis[j] + 1:
580                 lis[i] = lis[j]+1
581                 elements[i] = j
582     idx = 0
583
584     maximum = max(lis)
585     idx = lis.index(maximum)
586     seq = [myList[idx]]
587     while idx != elements[idx]:
588         idx = elements[idx]
589         seq.append(myList[idx])
590     return (maximum, reversed(seq))
591
592
593 myList = [10, 22, 9, 33, 21, 50, 41, 60]
594 ans = longest_increaing_subsequence(myList)
595 print('Length of lis is', ans[0])

```

```

596 print('The longest sequence is', ', '.join(str(x) for x in ans[1]))
597
598
599 **LONGEST CONTINUOUS ODD SUBSEQUENCE**
600
601
602 def longest_continuous_odd_subsequence(array):
603     final_list = []
604     temp_list = []
605     for i in array:
606         if i % 2 == 0:
607             if temp_list != []:
608                 final_list.append(temp_list)
609                 temp_list = []
610             else:
611                 temp_list.append(i)
612
613         if temp_list != []:
614             final_list.append(temp_list)
615     result = max(final_list, key=len)
616     print(result, len(result))
617
618
619 if __name__ == '__main__':
620     array = [2, 6, 8, 3, 9, 1, 5, 6, 1, 3, 5, 7, 7, 1, 2, 3, 4, 5]
621     longest_continuous_odd_subsequence(array)
622
623 **SIEVE OF ERATOSTHENES**
624
625
626 def sieve_of_eratosthenes(n):
627
628     prime = [True for i in range(n+1)]
629     p = 2
630     while (p * p <= n):
631         if (prime[p] == True):
632             for i in range(p * 2, n+1, p):
633                 prime[i] = False
634             p += 1
635     for p in range(2, n):
636         if prime[p]:
637             print(p),
638
639
640 if __name__ == '__main__':
641     n = 30
642     print("Following are the prime numbers smaller"),
643     print("than or equal to", n)
644     sieve_of_eratosthenes(n)
645
646 **GRAPH**
647
648
649 class AdjacencyList(object):
650     def __init__(self):
651         self.List = {}
652
653     def addEdge(self, fromVertex, toVertex):

```

```

654         if fromVertex in self.List.keys():
655             self.List[fromVertex].append(toVertex)
656         else:
657             self.List[fromVertex] = [toVertex]
658
659     def printList(self):
660         for i in self.List:
661             print(i, '->', ' -> '.join([str(j) for j in self.List[i]]))
662
663
664
665 if __name__ == '__main__':
666     a1 = AdjacencyList()
667     a1.addEdge(0, 1)
668     a1.addEdge(0, 4)
669     a1.addEdge(4, 1)
670     a1.addEdge(4, 3)
671     a1.addEdge(1, 0)
672     a1.addEdge(1, 4)
673     a1.addEdge(1, 3)
674     a1.addEdge(1, 2)
675     a1.addEdge(2, 3)
676     a1.addEdge(3, 4)
677     a1.printList()
678
679
680 **DEPTH FIRST SEARCH**
681
682
683 class Graph():
684     def __init__(self):
685         self.vertex = {}
686
687     def printGraph(self):
688         print(self.vertex)
689         for i in self.vertex.keys():
690             print(i, ' -> ', ' -> '.join([str(j) for j in
self.vertex[i]]))
691
692     def addEdge(self, fromVertex, toVertex):
693         if fromVertex in self.vertex.keys():
694             self.vertex[fromVertex].append(toVertex)
695         else:
696             self.vertex[fromVertex] = [toVertex]
697
698     def DFS(self):
699         visited = [False] * len(self.vertex)
700         for i in range(len(self.vertex)):
701             if visited[i] == False:
702                 self.DFSRec(i, visited)
703
704     def DFSRec(self, startVertex, visited):
705         visited[startVertex] = True
706         print(startVertex, end=' ')
707         for i in self.vertex.keys():
708             if visited[i] == False:
709                 self.DFSRec(i, visited)
710

```

```

711
712 if __name__ == '__main__':
713     g = Graph()
714     g.addEdge(0, 1)
715     g.addEdge(0, 2)
716     g.addEdge(1, 2)
717     g.addEdge(2, 0)
718     g.addEdge(2, 3)
719     g.addEdge(3, 3)
720     g.printGraph()
721     print('DFS:')
722     g.DFS()
723
724
725 **BREADTH FIRST SEARCH**
726
727
728 class Graph():
729     def __init__(self):
730         self.vertex = {}
731
732     def printGraph(self):
733         for i in self.vertex.keys():
734             print(i, ' -> ', ' -> '.join([str(j) for j in
self.vertex[i]]))
735
736     def addEdge(self, fromVertex, toVertex):
737         if fromVertex in self.vertex.keys():
738             self.vertex[fromVertex].append(toVertex)
739         else:
740             self.vertex[fromVertex] = [toVertex]
741
742     def BFS(self, startVertex):
743         visited = [False] * len(self.vertex)
744         queue = []
745         visited[startVertex] = True
746         queue.append(startVertex)
747         while queue:
748             startVertex = queue.pop(0)
749             print(startVertex, end=' ')
750             for i in self.vertex[startVertex]:
751                 if visited[i] == False:
752                     queue.append(i)
753                     visited[i] = True
754
755
756 if __name__ == '__main__':
757     g = Graph()
758     g.addEdge(0, 1)
759     g.addEdge(0, 2)
760     g.addEdge(1, 2)
761     g.addEdge(2, 0)
762     g.addEdge(2, 3)
763     g.addEdge(3, 3)
764     g.printGraph()
765     print('BFS:')
766     g.BFS(2)
767

```

```

768
769 **DETECT CYCLE IN DIRECTED GRAPH**
770
771
772 class Graph():
773     def __init__(self):
774         self.vertex = {}
775
776     def printGraph(self):
777         for i in self.vertex.keys():
778             print(i, ' -> ', ' -> '.join([str(j) for j in
self.vertex[i]]))
779
780     def addEdge(self, fromVertex, toVertex):
781         if fromVertex in self.vertex.keys():
782             self.vertex[fromVertex].append(toVertex)
783         else:
784             self.vertex[fromVertex] = [toVertex]
785
786     def checkCyclic(self):
787         visited = [False] * len(self.vertex)
788         stack = [False] * len(self.vertex)
789         for vertex in range(len(self.vertex)):
790             if visited[vertex] == False:
791                 if self.checkCyclicRec(visited, stack, vertex) == True:
792                     return True
793             return False
794
795     def checkCyclicRec(self, visited, stack, vertex):
796         visited[vertex] = True
797         stack[vertex] = True
798         for adjacentNode in self.vertex[vertex]:
799             if visited[adjacentNode] == False:
800                 if self.checkCyclicRec(visited, stack, adjacentNode) ==
True:
801                     return True
802                 elif stack[adjacentNode] == True:
803                     return True
804             stack[vertex] = False
805             return False
806
807
808 if __name__ == '__main__':
809     graph = Graph()
810     graph.addEdge(0, 1)
811     graph.addEdge(0, 2)
812     graph.addEdge(1, 2)
813     graph.addEdge(2, 0)
814     graph.addEdge(2, 3)
815     graph.addEdge(3, 3)
816     graph.printGraph()
817
818     if graph.checkCyclic() == 1:
819         print("Graph has a cycle")
820     else:
821         print("Graph has no cycle")
822
823

```



```

824  **DETECT CYCLE IN UNDIRECTED GRAPH**
825
826
827  class Graph():
828      def __init__(self):
829          self.vertex = {}
830
831      def printGraph(self):
832          for i in self.vertex.keys():
833              print(i, ' -> ', ' -> '.join([str(j) for j in
self.vertex[i]]))
834
835      def addEdge(self, fromVertex, toVertex):
836          if fromVertex in self.vertex.keys() and toVertex in
self.vertex.keys():
837              self.vertex[fromVertex].append(toVertex)
838              self.vertex[toVertex].append(fromVertex)
839          else:
840              self.vertex[fromVertex] = [toVertex]
841              self.vertex[toVertex] = [fromVertex]
842
843      def checkCyclic(self):
844          visited = [False] * len(self.vertex)
845          for vertex in range(len(self.vertex)):
846              if visited[vertex] == False:
847                  if self.checkCyclicRec(visited, -1, vertex) == True:
848                      return True
849          return False
850
851      def checkCyclicRec(self, visited, parent, vertex):
852          visited[vertex] = True
853          for adjacentNode in self.vertex[vertex]:
854              if visited[adjacentNode] == False:
855                  if self.checkCyclicRec(visited, vertex, adjacentNode) ==
True:
856                      return True
857                  elif parent != adjacentNode:
858                      return True
859          return False
860
861
862  if __name__ == '__main__':
863      graph = Graph()
864      graph.addEdge(0, 1)
865      graph.addEdge(0, 2)
866      graph.addEdge(1, 2)
867      graph.addEdge(2, 0)
868      graph.addEdge(2, 3)
869      graph.addEdge(3, 3)
870      graph.printGraph()
871
872      if graph.checkCyclic() == 1:
873          print("Graph has a cycle")
874      else:
875          print("Graph has no cycle")
876
877      g1 = Graph()
878      g1.addEdge(0, 1)

```

```

879         g1.addEdge(1, 2)
880         g1.printGraph()
881
882         if g1.checkCyclic() == 1:
883             print("Graph has a cycle")
884         else:
885             print("Graph has no cycle")
886
887
888     **TOPOLOGICAL SORT**
889
890
891     class Graph():
892         def __init__(self, count):
893             self.vertex = {}
894             self.count = count
895
896         def printGraph(self):
897             for i in self.vertex.keys():
898                 print(i, ' -> ', ' -> '.join([str(j) for j in
self.vertex[i]]))
899
900         def addEdge(self, fromVertex, toVertex):
901             if fromVertex in self.vertex.keys():
902                 self.vertex[fromVertex].append(toVertex)
903             else:
904                 self.vertex[fromVertex] = [toVertex]
905                 self.vertex[toVertex] = []
906
907         def topologicalSort(self):
908             visited = [False] * self.count
909             stack = []
910             for vertex in range(self.count):
911                 if visited[vertex] == False:
912                     self.topologicalSortRec(vertex, visited, stack)
913             print(' '.join([str(i) for i in stack]))
914
915         def topologicalSortRec(self, vertex, visited, stack):
916             visited[vertex] = True
917             try:
918                 for adjacentNode in self.vertex[vertex]:
919                     if visited[adjacentNode] == False:
920                         self.topologicalSortRec(adjacentNode, visited, stack)
921             except KeyError:
922                 return
923             stack.insert(0, vertex)
924
925
926     if __name__ == '__main__':
927         g = Graph(6)
928         g.addEdge(5, 2)
929         g.addEdge(5, 0)
930         g.addEdge(4, 0)
931         g.addEdge(4, 1)
932         g.addEdge(2, 3)
933         g.addEdge(3, 1)
934         g.topologicalSort()
935

```

```

936
937 **PRIM'S ALGORITHM**
938
939
940 class Graph(object):
941     def __init__(self, vertices):
942         self.vertices = vertices
943         self.graph = [[0 for column in range(vertices)]
944                        for row in range(vertices)]
945
946     def getMinimumKey(self, weight, visited):
947         min = 9999
948         for i in range(self.vertices):
949             if weight[i] < min and visited[i] == False:
950                 min = weight[i]
951                 minIndex = i
952         return minIndex
953
954     def primsAlgo(self):
955         weight = [9999] * self.vertices
956         MST = [None] * self.vertices
957         weight[0] = 0
958         visited = [False] * self.vertices
959         MST[0] = -1
960         for _ in range(self.vertices):
961             minIndex = self.getMinimumKey(weight, visited)
962             visited[minIndex] = True
963             for vertex in range(self.vertices):
964                 if self.graph[minIndex][vertex] > 0 and visited[vertex] ==
False and \
965                     weight[vertex] > self.graph[minIndex][vertex]:
966                     weight[vertex] = self.graph[minIndex][vertex]
967                     MST[vertex] = minIndex
968             self.printMST(MST)
969
970     def printMST(self, MST):
971         print("Edge \tweight")
972         for i in range(1, self.vertices):
973             print(MST[i], "-", i, "\t", self.graph[i][MST[i]])
974
975
976 if __name__ == '__main__':
977     g = Graph(6)
978
979     g.graph = [[0, 3, 2, 5, 7, 3],
980                [3, 0, 4, 8, 6, 6],
981                [2, 4, 0, 7, 1, 3],
982                [5, 8, 7, 0, 2, 4],
983                [7, 6, 1, 2, 0, 3],
984                [3, 6, 3, 4, 3, 0]]
985
986     g.primsAlgo()
987
988
989 **KRUSKAL'S ALGORITHM**
990
991
992 class Graph:

```

```

993     def __init__(self, vertices):
994         self.vertices = vertices
995         self.graph = []
996
997     def addEdge(self, fromEdge, toEdge, weight):
998         self.graph.append([fromEdge, toEdge, weight])
999
1000    def find(self, parent, i):
1001        if parent[i] == i:
1002            return i
1003        return self.find(parent, parent[i])
1004
1005    def union(self, parent, rank, first, second):
1006        root_x = self.find(parent, first)
1007        root_y = self.find(parent, second)
1008
1009        if rank[root_x] < rank[root_y]:
1010            parent[root_x] = root_y
1011        elif rank[root_x] > rank[root_y]:
1012            parent[root_y] = root_x
1013
1014        elif rank[root_x] == rank[root_y]:
1015            parent[root_y] = root_x
1016            rank[root_x] += 1
1017
1018    def kruskals(self):
1019        result = []
1020        sortedIndex = 0
1021        resultIndex = 0
1022        self.graph = sorted(self.graph, key=lambda item: item[2])
1023        parent = []
1024        rank = []
1025        for node in range(self.vertices):
1026            parent.append(node)
1027            rank.append(0)
1028
1029        while resultIndex < self.vertices - 1:
1030            fromEdge, toEdge, weight = self.graph[sortedIndex]
1031            sortedIndex += 1
1032            root_x = self.find(parent, fromEdge)
1033            root_y = self.find(parent, toEdge)
1034            if root_x != root_y:
1035                resultIndex += 1
1036                result.append([fromEdge, toEdge, weight])
1037                self.union(parent, rank, root_x, root_y)
1038
1039        print('Constructed Kruskal\'s Minimum Spanning Tree: ')
1040        for u, v, weight in result:
1041            print('{} -> {} = {}'.format(u, v, weight))
1042
1043
1044    if __name__ == '__main__':
1045        g = Graph(4)
1046        g.addEdge(0, 1, 10)
1047        g.addEdge(0, 2, 6)
1048        g.addEdge(0, 3, 5)
1049        g.addEdge(1, 3, 15)
1050        g.addEdge(2, 3, 4)

```

```

1051     g.kruskals()
1052
1053     **HEAP**
1054
1055
1056     class BinaryHeap(object):
1057         def __init__(self):
1058             self.heap = [0]
1059             self.currentSize = 0
1060
1061         def __repr__(self):
1062             heap = self.heap[1:]
1063             return ' '.join(str(i) for i in heap)
1064
1065         def shiftUp(self, index):
1066             while (index // 2) > 0:
1067                 if self.heap[index] < self.heap[index // 2]:
1068                     temp = self.heap[index // 2]
1069                     self.heap[index // 2] = self.heap[index]
1070                     self.heap[index] = temp
1071                 index = index // 2
1072
1073         def insert(self, key):
1074             self.heap.append(key)
1075             self.currentSize += 1
1076             self.shiftUp(self.currentSize)
1077
1078         def shiftDown(self, index):
1079             while (index * 2) <= self.currentSize:
1080                 minimumChild = self.minChild(index)
1081                 if self.heap[index] > self.heap[minimumChild]:
1082                     temp = self.heap[index]
1083                     self.heap[index] = self.heap[minimumChild]
1084                     self.heap[minimumChild] = temp
1085                 index = minimumChild
1086
1087         def minChild(self, i):
1088             if i * 2 + 1 > self.currentSize:
1089                 return i * 2
1090             else:
1091                 if self.heap[i * 2] < self.heap[i * 2 + 1]:
1092                     return i * 2
1093                 else:
1094                     return i * 2 + 1
1095
1096         def delete(self):
1097             deletedNode = self.heap[1]
1098             self.heap[1] = self.heap[self.currentSize]
1099             self.currentSize -= 1
1100             self.heap.pop()
1101             self.shiftDown(1)
1102             return deletedNode
1103
1104         def buildHeap(self, alist):
1105             i = len(alist) // 2
1106             self.currentSize = len(alist)
1107             self.heap = [0] + alist[:]
1108             while (i > 0):

```

```

1109         self.shiftDown(i)
1110         i = i - 1
1111
1112
1113 bh = BinaryHeap()
1114 bh.buildHeap([9, 5, 6, 2, 3])
1115
1116 print('Deleted:', bh.delete())
1117 print('Deleted:', bh.delete())
1118 print('Deleted:', bh.delete())
1119 bh.insert(3)
1120 print('Deleted:', bh.delete())
1121 print(bh)
1122
1123
1124 **HEAP SORT**
1125
1126
1127 def HeapSort(alist):
1128     heapify(alist)
1129     end = len(alist) - 1
1130     while end > 0:
1131         alist[end], alist[0] = alist[0], alist[end]
1132         shiftDown(alist, 0, end - 1)
1133         end -= 1
1134
1135
1136 def heapify(alist):
1137     ''' This function helps to maintain the heap property '''
1138     start = len(alist) // 2
1139     while start >= 0:
1140         shiftDown(alist, start, len(alist) - 1)
1141         start -= 1
1142
1143
1144 def shiftDown(alist, start, end):
1145     root = start
1146     while root * 2 + 1 <= end:
1147         child = root * 2 + 1
1148
1149         if child + 1 <= end and alist[child] < alist[child + 1]:
1150             child += 1
1151
1152         if child <= end and alist[root] < alist[child]:
1153             alist[root], alist[child] = alist[child], alist[root]
1154             root = child
1155     else:
1156         return
1157
1158
1159 if __name__ == '__main__':
1160     alist = [12, 2, 4, 5, 2, 3]
1161     HeapSort(alist)
1162     print('Sorted Array:', alist)
1163
1164 **MAX HEAP**
1165
1166

```

```

1167 def heapify(A):
1168     '''Turns a list `A` into a max-ordered binary heap.'''
1169     n = len(A) - 1
1170     for node in range(n/2, -1, -1):
1171         __shift_down(A, node)
1172     return
1173
1174
1175 def push_heap(A, val):
1176     '''Pushes a value onto the heap `A` while keeping the heap property
1177     intact. The heap size increases by 1.'''
1178     A.append(val)
1179     __shift_up(A, len(A) - 1)
1180     return
1181
1182
1183 def pop_heap(A):
1184     '''Returns the max value from the heap `A` while keeping the heap
1185     property intact. The heap size decreases by 1.'''
1186     n = len(A) - 1
1187     __swap(A, 0, n)
1188     max = A.pop(n)
1189     __shift_down(A, 0)
1190     return max
1191
1192
1193 def replace_key(A, node, newval):
1194     '''Replace the key at node `node` in the max-heap `A` by `newval`.
1195     The heap size does not change.'''
1196     curval = A[node]
1197     A[node] = newval
1198
1199     if newval > curval:
1200         __shift_up(A, node)
1201
1202     elif newval < curval:
1203         __shift_down(A, node)
1204     return
1205
1206
1207 def __swap(A, i, j):
1208     A[i], A[j] = A[j], A[i]
1209     return
1210
1211
1212 def __shift_down(A, node):
1213     '''Traverse down a binary tree `A` starting at node `node` and
1214     turn it into a max-heap'''
1215     child = 2*node + 1
1216
1217     if child > len(A) - 1:
1218         return
1219
1220     if (child + 1 <= len(A) - 1) and (A[child+1] > A[child]):
1221         child += 1
1222
1223     if A[node] < A[child]:
1224         __swap(A, node, child)

```

```

1225         __shiftdown(A, child)
1226     else:
1227         return
1228
1229
1230 def __shiftup(A, node):
1231     '''Traverse up an otherwise max-heap `A` starting at node `node`
1232     (which is the only node that breaks the heap property) and restore
1233     the heap structure.'''
1234     parent = (node - 1)/2
1235     if A[parent] < A[node]:
1236         __swap(A, node, parent)
1237
1238     if parent <= 0:
1239         return
1240     else:
1241         __shiftup(A, parent)
1242
1243
1244 **SINGLY LINKED LIST**
1245
1246
1247 class Node(object):
1248
1249     def __init__(self, data, next=None):
1250         self.data = data
1251         self.next = next
1252
1253     def setData(self, data):
1254         self.data = data
1255
1256     def getData(self):
1257         return self.data
1258
1259     def setNext(self, next):
1260         self.next = next
1261
1262     def getNext(self):
1263         return self.next
1264
1265
1266 class LinkedList(object):
1267     def __init__(self):
1268         self.head = None
1269
1270     def printLinkedList(self):
1271         temp = self.head
1272         while(temp):
1273             print(temp.data, end=' ')
1274             temp = temp.next
1275
1276     def insertAtStart(self, data):
1277         if self.head == None:
1278             newNode = Node(data)
1279             self.head = newNode
1280         else:
1281             newNode = Node(data)
1282             newNode.next = self.head

```



```

1283         self.head = newNode
1284
1285     def insertBetween(self, previousNode, data):
1286         if (previousNode.next is None):
1287             print('Previous node should have next node!')
1288         else:
1289             newNode = Node(data)
1290             newNode.next = previousNode.next
1291             previousNode.next = newNode
1292
1293     def insertAtEnd(self, data):
1294         newNode = Node(data)
1295         temp = self.head
1296         while(temp.next != None):
1297             temp = temp.next
1298         temp.next = newNode
1299
1300     def delete(self, data):
1301         temp = self.head
1302         if (temp.next is not None):
1303             if(temp.data == data):
1304                 self.head = temp.next
1305                 temp = None
1306                 return
1307             else:
1308                 while(temp.next != None):
1309                     if(temp.data == data):
1310                         break
1311                     prev = temp
1312                     temp = temp.next
1313                 if temp == None:
1314                     return
1315                 prev.next = temp.next
1316                 return
1317
1318     def search(self, node, data):
1319         if node == None:
1320             return False
1321         if node.data == data:
1322             return True
1323         return self.search(node.getNext(), data)
1324
1325
1326 if __name__ == '__main__':
1327     List = LinkedList()
1328     List.head = Node(1)
1329     node2 = Node(2)
1330     List.head.setNext(node2)
1331     node3 = Node(3)
1332     node2.setNext(node3)
1333     List.insertAtStart(4)
1334     List.insertBetween(node2, 5)
1335     List.insertAtEnd(6)
1336     List.printLinkedList()
1337     print()
1338     List.delete(3)
1339     List.printLinkedList()
1340     print()

```

```

1341     print(List.search(List.head, 1))
1342
1343 **CIRCULAR LINKED LIST**
1344
1345
1346 class Node:
1347     def __init__(self, data):
1348         self.data = data
1349         self.next = None
1350
1351
1352 class CreateList:
1353     def __init__(self):
1354         self.head = Node(None)
1355         self.tail = Node(None)
1356         self.head.next = self.tail
1357         self.tail.next = self.head
1358
1359     def add(self, data):
1360         newNode = Node(data)
1361         if self.head.data is None:
1362             self.head = newNode
1363             self.tail = newNode
1364             newNode.next = self.head
1365         else:
1366             self.tail.next = newNode
1367             self.tail = newNode
1368             self.tail.next = self.head
1369
1370     def display(self):
1371         current = self.head
1372         if self.head is None:
1373             print("List is empty")
1374             return
1375         else:
1376             print("Nodes of the circular linked list: ")
1377             print(current.data),
1378             while(current.next != self.head):
1379                 current = current.next
1380                 print(current.data),
1381
1382
1383 class CircularLinkedList:
1384     cl = CreateList()
1385     cl.add(1)
1386     cl.add(2)
1387     cl.add(3)
1388     cl.add(4)
1389     cl.display()
1390
1391
1392 **DOUBLY LINKED LIST**
1393
1394
1395 class Node(object):
1396     def __init__(self, data, next=None, previous=None):
1397         self.data = data
1398         self.next = next

```

```

1399         self.previous = previous
1400
1401
1402 class DoublyLinkedList(object):
1403     def __init__(self):
1404         self.head = None
1405
1406     def insertAtStart(self, data):
1407         if self.head == None:
1408             newNode = Node(data)
1409             self.head = newNode
1410         else:
1411             newNode = Node(data)
1412             self.head.previous = newNode
1413             newNode.next = self.head
1414             self.head = newNode
1415
1416     def insertAtEnd(self, data):
1417         newNode = Node(data)
1418         temp = self.head
1419         while(temp.next != None):
1420             temp = temp.next
1421         temp.next = newNode
1422         newNode.previous = temp
1423
1424     def delete(self, data):
1425         temp = self.head
1426         if(temp.next != None):
1427
1428             if(temp.data == data):
1429                 temp.next.previous = None
1430                 self.head = temp.next
1431                 temp.next = None
1432                 return
1433             else:
1434                 while(temp.next != None):
1435                     if(temp.data == data):
1436                         break
1437                     temp = temp.next
1438                 if(temp.next):
1439
1440                     temp.previous.next = temp.next
1441                     temp.next.previous = temp.previous
1442                     temp.next = None
1443                     temp.previous = None
1444                 else:
1445
1446                     temp.previous.next = None
1447                     temp.previous = None
1448                 return
1449
1450         if (temp == None):
1451             return
1452
1453     def printdll(self):
1454         temp = self.head
1455         while(temp != None):
1456             print(temp.data, end=' ')

```

```

1457         temp = temp.next
1458
1459
1460 if __name__ == '__main__':
1461     dll = DoublyLinkedList()
1462     dll.insertAtStart(1)
1463     dll.insertAtStart(2)
1464     dll.insertAtEnd(3)
1465     dll.insertAtStart(4)
1466     dll.printdll()
1467     dll.delete(2)
1468     print()
1469     dll.printdll()
1470
1471 **LENGTH OF LINKED LIST**
1472
1473
1474 def checkLength(linkedList):
1475     count = 0
1476     temp = linkedList.head
1477     while(temp != None):
1478         count += 1
1479         temp = temp.next
1480     return count
1481
1482
1483 if __name__ == '__main__':
1484     myLinkedList = SinglyLinkedList.LinkedList()
1485     for i in range(10):
1486         myLinkedList.insertAtStart(i)
1487     myLinkedList.printLinkedList()
1488     print()
1489     print('Length:', checkLength(myLinkedList))
1490
1491
1492 **REVERSING LINKED LIST**
1493
1494
1495 def reverseLinkedList(myLinkedList):
1496     previous = None
1497     current = myLinkedList.head
1498     while(current != None):
1499         temp = current.next
1500         current.next = previous
1501         previous = current
1502         current = temp
1503     myLinkedList.head = previous
1504
1505
1506 if __name__ == '__main__':
1507     myLinkedList = SinglyLinkedList.LinkedList()
1508     for i in range(10, 0, -1):
1509         myLinkedList.insertAtStart(i)
1510     print('Original:', end=' ')
1511     myLinkedList.printLinkedList()
1512     print()
1513     print('Reversed:', end=' ')
1514     reverseLinkedList(myLinkedList)

```

```
1515         myLinkedList.printLinkedList()
1516
1517
1518     **QUEUE**
1519
1520
1521     class Queue(object):
1522         def __init__(self, limit=10):
1523             self.queue = []
1524             self.front = None
1525             self.rear = None
1526             self.limit = limit
1527             self.size = 0
1528
1529         def __str__(self):
1530             return ' '.join([str(i) for i in self.queue])
1531
1532         def isEmpty(self):
1533             return self.size <= 0
1534
1535         def enqueue(self, data):
1536             if self.size >= self.limit:
1537                 return -1
1538             else:
1539                 self.queue.append(data)
1540                 if self.front is None:
1541                     self.front = self.rear = 0
1542                 else:
1543                     self.rear = self.size
1544                 self.size += 1
1545
1546         def dequeue(self):
1547             if self.isEmpty():
1548                 return -1
1549             else:
1550                 self.queue.pop(0)
1551                 self.size -= 1
1552                 if self.size == 0:
1553                     self.front = self.rear = 0
1554                 else:
1555                     self.rear = self.size - 1
1556
1557         def getSize(self):
1558             return self.size
1559
1560
1561     if __name__ == '__main__':
1562         myQueue = Queue()
1563         for i in range(10):
1564             myQueue.enqueue(i)
1565         print(myQueue)
1566         print('Queue Size:', myQueue.getSize())
1567         myQueue.dequeue()
1568         print(myQueue)
1569         print('Queue Size:', myQueue.getSize())
1570
1571     **DEQUE**
1572
```

```

1573
1574 class Deque(object):
1575     def __init__(self, limit=10):
1576         self.queue = []
1577         self.limit = limit
1578
1579     def __str__(self):
1580         return ' '.join([str(i) for i in self.queue])
1581
1582     def isEmpty(self):
1583         return len(self.queue) <= 0
1584
1585     def isFull(self):
1586         return len(self.queue) >= self.limit
1587
1588     def insertRear(self, data):
1589         if self.isFull():
1590             return
1591         else:
1592             self.queue.insert(0, data)
1593
1594     def insertFront(self, data):
1595         if self.isFull():
1596             return
1597         else:
1598             self.queue.append(data)
1599
1600     def deleteRear(self):
1601         if self.isEmpty():
1602             return
1603         else:
1604             return self.queue.pop(0)
1605
1606     def deleteFront(self):
1607         if self.isFull():
1608             return
1609         else:
1610             return self.queue.pop()
1611
1612
1613 if __name__ == '__main__':
1614     myDeque = Deque()
1615     myDeque.insertFront(1)
1616     myDeque.insertRear(2)
1617     myDeque.insertFront(3)
1618     myDeque.insertRear(10)
1619     print(myDeque)
1620     myDeque.deleteRear()
1621     print(myDeque)
1622     myDeque.deleteFront()
1623     print(myDeque)
1624
1625 **CIRCULAR QUEUE**
1626
1627
1628 class CircularQueue(object):
1629     def __init__(self, limit=10):
1630         self.limit = limit

```

```

1631         self.queue = [None for i in range(limit)]
1632         self.front = self.rear = -1
1633
1634     def __str__(self):
1635         if (self.rear >= self.front):
1636             return ' '.join([str(self.queue[i]) for i in range(self.front,
self.rear + 1)])
1637
1638         else:
1639             q1 = ' '.join([str(self.queue[i])
for i in range(self.front, self.limit)])
1640             q2 = ' '.join([str(self.queue[i])
for i in range(0, self.rear + 1)])
1641             return q1 + ' ' + q2
1642
1643     def isEmpty(self):
1644         return self.front == -1
1645
1646     def isFull(self):
1647         return (self.rear + 1) % self.limit == self.front
1648
1649     def enqueue(self, data):
1650         if self.isFull():
1651             print('Queue is Full!')
1652         elif self.isEmpty():
1653             self.front = 0
1654             self.rear = 0
1655             self.queue[self.rear] = data
1656         else:
1657             self.rear = (self.rear + 1) % self.limit
1658             self.queue[self.rear] = data
1659
1660     def dequeue(self):
1661         if self.isEmpty():
1662             print('Queue is Empty!')
1663         elif (self.front == self.rear):
1664             self.front = -1
1665             self.rear = -1
1666         else:
1667             self.front = (self.front + 1) % self.limit
1668
1669 if __name__ == '__main__':
1670     myCQ = CircularQueue(5)
1671     myCQ.enqueue(14)
1672     myCQ.enqueue(22)
1673     myCQ.enqueue(13)
1674     myCQ.enqueue(16)
1675     print('Queue:', myCQ)
1676     myCQ.dequeue()
1677     myCQ.dequeue()
1678     print('Queue:', myCQ)
1679     myCQ.enqueue(9)
1680     myCQ.enqueue(20)
1681     myCQ.enqueue(5)
1682     myCQ.enqueue(5)
1683     print('Queue:', myCQ)
1684

```

```

1688  **PRIORITY QUEUE**
1689
1690
1691  class PriorityQueue(object):
1692      def __init__(self):
1693          self.queue = []
1694
1695      def __str__(self):
1696          return ' '.join([str(i) for i in self.queue])
1697
1698      def isEmpty(self):
1699          return len(self.queue) == []
1700
1701      def insert(self, data):
1702          self.queue.append(data)
1703
1704      def delete(self):
1705          try:
1706              max = 0
1707              for i in range(len(self.queue)):
1708                  if self.queue[i] > self.queue[max]:
1709                      max = i
1710              item = self.queue[max]
1711              del self.queue[max]
1712              return item
1713          except IndexError:
1714              print()
1715              exit()
1716
1717
1718  if __name__ == '__main__':
1719      myQueue = PriorityQueue()
1720      myQueue.insert(12)
1721      myQueue.insert(1)
1722      myQueue.insert(14)
1723      myQueue.insert(7)
1724      print(myQueue)
1725      while not myQueue.isEmpty():
1726          print(myQueue.delete(), end=' ')
1727
1728  **SEGMENT TREE**
1729
1730
1731  class SegmentTree:
1732      def __init__(self, values):
1733          self.valarr = values
1734          self.arr = dict()
1735
1736      def buildTree(self, start, end, node):
1737          if start == end:
1738              self.arr[node] = self.valarr[start]
1739              return
1740          mid = (start+end)//2
1741          self.buildTree(start, mid, node*2)
1742          self.buildTree(mid+1, end, node*2+1)
1743          self.arr[node] = self.arr[node*2]+self.arr[node*2+1]
1744
1745      def rangeQuery(self, node, start, end, l, r):

```



```

1746         if (l <= start and r >= end):
1747             return self.arr[node]
1748
1749         if (end < l or start > r):
1750             return 0
1751
1752         mid = (start+end)//2
1753         return self.rangeQuery(2*node, start, mid, l, r) +
1754 self.rangeQuery(2*node+1, mid+1, end, l, r)
1755
1756     def update(self, node, newvalue, oldvalue, position, start, end):
1757
1758         if start <= position <= end:
1759             self.arr[node] += (newvalue-oldvalue)
1760
1761         if start != end:
1762             mid = (start+end)//2
1763             self.update(node*2, newvalue, oldvalue, position, start, mid)
1764             self.update(node*2+1, newvalue, oldvalue, position, mid+1,
1765 end)
1766
1767 if __name__ == '__main__':
1768     l = list(
1769         map(int, input("Enter the elements of the array separated by
1770 space:\n").split()))
1771     st = SegmentTree(l)
1772     st.buildTree(0, len(l)-1, 1)
1773     baseindex = 1
1774     endindex = len(l)
1775     print(st.arr)
1776     print("Sum of numbers from index 3 and 5 is: ",
1777           st.rangeQuery(1, baseindex, endindex, 3, 5))
1778     updateindex = 3
1779     updatevalue = 10
1780     st.update(1, updatevalue, l[updateindex-1],
1781               updateindex, baseindex, endindex)
1782
1783     print("Updated sum of numbers from index 3 and 5 is: ",
1784           st.rangeQuery(1, baseindex, endindex, 3, 5))
1785
1786 **SEGMENT TREE 2**
1787
1788 class SegmentTree:
1789     def __init__(self, values):
1790         self.minarr = dict()
1791
1792         self.originalarr = values[:]
1793
1794     def buildminTree(self, start, end, node):
1795         if start == end:
1796             self.minarr[node] = self.originalarr[start]
1797             return
1798         mid = (start + end) // 2
1799         self.buildminTree(start, mid, node*2)
1800         self.buildminTree(mid + 1, end, node*2 + 1)

```

```

1801         self.minarr[node] = min(self.minarr[node*2], self.minarr[node*2 +
1802 1])
1803
1804     def minRangeQuery(self, node, start, end, l, r):
1805         if l <= start and end <= r:
1806             return self.minarr[node]
1807         if r < start or l > end:
1808             return sys.maxsize
1809         mid = (start+end)//2
1810         return min(self.minRangeQuery(node*2, start, mid, l, r),
1811 self.minRangeQuery(node*2+1, mid+1, end, l, r))
1812
1813     def update(self, node, newvalue, position, start, end):
1814         if start <= position <= end:
1815             self.minarr[node] = min(self.minarr[node], newvalue)
1816
1817         if start != end:
1818             mid = (start + end) // 2
1819             self.update(node * 2, newvalue, position, start, mid)
1820             self.update(node * 2 + 1, newvalue, position, mid + 1, end)
1821
1822 arr = [10, 5, 9, 3, 4, 8, 6, 7, 2, 1]
1823 st = SegmentTree(arr)
1824 st.buildminTree(0, len(arr)-1, 1)
1825 print("Segment Tree for given array", st.minarr)
1826 print("Minimum of numbers from index 6 to 9 is: ",
1827 st.minRangeQuery(1, 1, len(arr), 6, 9))
1828 st.update(1, 2, 4, 1, len(arr))
1829 print(st.minarr)
1830 print("Updated minimum of numbers from index 2 to 9 is: ",
1831 st.minRangeQuery(1, 1, len(arr), 2, 6))
1832
1833 **STACK**
1834
1835
1836 class Stack(object):
1837     def __init__(self, limit=10):
1838         self.stack = []
1839         self.limit = limit
1840
1841     def __str__(self):
1842         return ' '.join([str(i) for i in self.stack])
1843
1844     def push(self, data):
1845         if len(self.stack) >= self.limit:
1846             print('Stack overflow')
1847         else:
1848             self.stack.append(data)
1849
1850     def pop(self):
1851         if len(self.stack) <= 0:
1852             return -1
1853         else:
1854             return self.stack.pop()
1855
1856     def peek(self):

```

```

1857         if len(self.stack) <= 0:
1858             return -1
1859         else:
1860             return self.stack[len(self.stack) - 1]
1861
1862     def isEmpty(self):
1863         return self.stack == []
1864
1865     def size(self):
1866         return len(self.stack)
1867
1868
1869 if __name__ == '__main__':
1870     myStack = Stack()
1871     for i in range(10):
1872         myStack.push(i)
1873     print(myStack)
1874     myStack.pop()
1875     print(myStack)
1876     myStack.peek()
1877     myStack.isEmpty()
1878     myStack.size()
1879
1880 **INFIX TO POSTFIX**
1881
1882
1883 def isOperand(char):
1884     return (ord(char) >= ord('a') and ord(char) <= ord('z')) or (ord(char)
1885 >= ord('A') and ord(char) <= ord('Z'))
1886
1887
1888 def precedence(char):
1889     if char == '+' or char == '-':
1890         return 1
1891     elif char == '*' or char == '/':
1892         return 2
1893     elif char == '^':
1894         return 3
1895     else:
1896         return -1
1897
1898
1899 def infixToPostfix(myExp, myStack):
1900     postFix = []
1901     for i in range(len(myExp)):
1902         if (isOperand(myExp[i])):
1903             postFix.append(myExp[i])
1904         elif(myExp[i] == '('):
1905             myStack.push(myExp[i])
1906         elif(myExp[i] == ')'):
1907             topOperator = myStack.pop()
1908             while(not myStack.isEmpty() and topOperator != '('):
1909                 postFix.append(topOperator)
1910                 topOperator = myStack.pop()
1911             else:
1912                 while (not myStack.isEmpty()) and (precedence(myExp[i]) <=
precedence(myStack.peek())):
1913                     postFix.append(myStack.pop())

```

```

1913         myStack.push(myExp[i])
1914
1915     while(not myStack.isEmpty()):
1916         postFix.append(myStack.pop())
1917     return ' '.join(postFix)
1918
1919
1920 if __name__ == '__main__':
1921     myExp = 'a+b*(c^d-e)^(f+g*h)-i'
1922     myExp = [i for i in myExp]
1923     print('Infix:', ' '.join(myExp))
1924     myStack = Stack.Stack(len(myExp))
1925     print('Postfix:', infixToPostfix(myExp, myStack))
1926
1927
1928 **BALANCED PARANTHESIS**
1929
1930
1931 def parseParenthesis(string):
1932     balanced = 1
1933     index = 0
1934     myStack = Stack.Stack(len(string))
1935     while (index < len(string)) and (balanced == 1):
1936         check = string[index]
1937         if check == '(':
1938             myStack.push(check)
1939         else:
1940             if myStack.isEmpty():
1941                 balanced = 0
1942             else:
1943                 myStack.pop()
1944         index += 1
1945
1946     if balanced == 1 and myStack.isEmpty():
1947         return True
1948     else:
1949         return False
1950
1951
1952 if __name__ == '__main__':
1953     print(parseParenthesis('((( )))'))
1954     print(parseParenthesis('(( ( ) )'))
1955
1956 **DECIMAL TO BINARY**
1957
1958
1959 def dtob(decimal, base=2):
1960     myStack = Stack.Stack()
1961     while decimal > 0:
1962         myStack.push(decimal % base)
1963         decimal //= base
1964
1965     result = ''
1966     while not myStack.isEmpty():
1967         result += str(myStack.pop())
1968
1969     return result
1970

```

```
1971
1972 if __name__ == '__main__':
1973     print(dtob(15))
1974
1975 **REVERSE STRING**
1976
1977
1978 def reverse(string):
1979     myStack = Stack.Stack(len(string))
1980     for i in string:
1981         myStack.push(i)
1982     result = ''
1983     while not myStack.isEmpty():
1984         result += myStack.pop()
1985     return result
1986
1987
1988 if __name__ == '__main__':
1989     print(reverse('omkar'))
1990
1991 **QUEUE IMPLEMENTATION USING TWO STACKS**
1992
1993
1994 class StackedQueue:
1995
1996     def __init__(self):
1997         self.stack = Stack()
1998         self.alternateStack = Stack()
1999
2000     def enqueue(self, item):
2001         while(not self.stack.is_empty()):
2002             self.alternateStack.push(self.stack.pop())
2003         self.alternateStack.push(item)
2004         while(not self.alternateStack.is_empty()):
2005             self.stack.push(self.alternateStack.pop())
2006
2007     def dequeue(self):
2008         return self.stack.pop()
2009
2010     def __repr__(self):
2011         return repr(self.stack)
2012
2013
2014 class Stack:
2015     def __init__(self):
2016         self.items = []
2017
2018     def push(self, item):
2019         self.items.append(item)
2020
2021     def pop(self):
2022         return self.items.pop()
2023
2024     def size(self):
2025         return len(self.items)
2026
2027     def is_empty(self):
2028         return self.items == []
```

```
2029
2030     def __repr__(self):
2031         return str(self.items)
2032
2033
2034 if __name__ == "__main__":
2035     structure = StackedQueue()
2036     structure.enqueue(4)
2037     structure.enqueue(3)
2038     structure.enqueue(2)
2039     structure.enqueue(1)
2040     print(structure)
2041     structure.dequeue()
2042     print(structure)
2043
2044 **TREE**
2045
2046
2047 class Node(object):
2048     def __init__(self, data=None):
2049         self.left = None
2050         self.right = None
2051         self.data = data
2052
2053     def setLeft(self, node):
2054         self.left = node
2055
2056     def setRight(self, node):
2057         self.right = node
2058
2059     def getLeft(self):
2060         return self.left
2061
2062     def getRight(self):
2063         return self.right
2064
2065     def setData(self, data):
2066         self.data = data
2067
2068     def getData(self):
2069         return self.data
2070
2071
2072 def inorder(Tree):
2073     if Tree:
2074         inorder(Tree.getLeft())
2075         print(Tree.getData(), end=' ')
2076         inorder(Tree.getRight())
2077     return
2078
2079
2080 def preorder(Tree):
2081     if Tree:
2082         print(Tree.getData(), end=' ')
2083         preorder(Tree.getLeft())
2084         preorder(Tree.getRight())
2085     return
2086
```

```

2087
2088 def postorder(Tree):
2089     if Tree:
2090         postorder(Tree.getLeft())
2091         postorder(Tree.getRight())
2092         print(Tree.getData(), end=' ')
2093     return
2094
2095
2096 if __name__ == '__main__':
2097     root = Node(1)
2098     root.setLeft(Node(2))
2099     root.setRight(Node(3))
2100     root.left.setLeft(Node(4))
2101     print('Inorder Traversal:')
2102     inorder(root)
2103     print('\nPreorder Traversal:')
2104     preorder(root)
2105     print('\nPostorder Traversal:')
2106     postorder(root)
2107
2108
2109 **TREE TRAVERSAL**
2110
2111
2112 class Node(object):
2113     def __init__(self, val):
2114         self.key = val
2115         self.left = None
2116         self.right = None
2117
2118
2119 class Tree:
2120     def __init__(self, val):
2121         self.root = Node(val)
2122
2123     def insertNode(root, val):
2124         if(root == None):
2125             root = Node(val)
2126         elif(root.key < val):
2127             root.right = Tree.insertNode(root.right, val)
2128         else:
2129             root.left = Tree.insertNode(root.left, val)
2130         return root
2131
2132     def inorder(root):
2133         if(root == None):
2134             return ""
2135         else:
2136             return str(Tree.inorder(root.left)) + " " + str(root.key) + "
2137 " + str(Tree.inorder(root.right))
2138
2139     def preorder(root):
2140         if(root == None):
2141             return ""
2142         else:
2143             return str(root.key) + " " + str(Tree.preorder(root.left)) + "
2144 " + str(Tree.preorder(root.right))

```

```

2143
2144     def postorder(root):
2145         if(root == None):
2146             return ""
2147         else:
2148             return str(Tree.postorder(root.left)) + " " +
str(Tree.postorder(root.right)) + " " + str(root.key)
2149
2150
2151 array = [1, 22, 3, 44, 32, 35]
2152 treeRoot = Node(array[0])
2153 for i in range(1, len(array)):
2154     treeRoot = Tree.insertNode(treeRoot, array[i])
2155 print("Inorder:", Tree.inorder(treeRoot))
2156 print("Preorder:", Tree.preorder(treeRoot))
2157 print("Postorder:", Tree.postorder(treeRoot))
2158
2159
2160 **ZIGZAG TRAVERSAL**
2161
2162
2163 class Node:
2164
2165     def __init__(self, data):
2166         self.left = None
2167         self.right = None
2168         self.data = data
2169
2170
2171 def make_tree() -> Node:
2172     root = Node(1)
2173     root.left = Node(2)
2174     root.right = Node(3)
2175     root.left.left = Node(4)
2176     root.left.right = Node(5)
2177     return root
2178
2179
2180 def zigzag_iterative(root: Node):
2181     if root == None:
2182         return
2183     s1 = []
2184     s2 = []
2185     s1.append(root)
2186     while not len(s1) == 0 or not len(s2) == 0:
2187         while not len(s1) == 0:
2188             temp = s1[-1]
2189             s1.pop()
2190             print(temp.data, end=" ")
2191             if temp.left:
2192                 s2.append(temp.left)
2193             if temp.right:
2194                 s2.append(temp.right)
2195
2196         while not len(s2) == 0:
2197             temp = s2[-1]
2198             s2.pop()
2199             print(temp.data, end=" ")

```



```

2200         if temp.right:
2201             s1.append(temp.right)
2202         if temp.left:
2203             s1.append(temp.left)
2204
2205
2206 def main():
2207     root = make_tree()
2208     print("\nzigzag order traversal(iterative) is: ")
2209     zigzag_iterative(root)
2210     print()
2211
2212
2213 if __name__ == "__main__":
2214     import doctest
2215     doctest.testmod()
2216     main()
2217
2218
2219 **BINARY SEARCH TREE**
2220
2221
2222 class Node(object):
2223     def __init__(self, data):
2224         self.data = data
2225         self.leftChild = None
2226         self.rightChild = None
2227
2228     def insert(self, data):
2229         if self.data == data:
2230             return False
2231
2232         elif data < self.data:
2233             if self.leftChild:
2234                 return self.leftChild.insert(data)
2235             else:
2236                 self.leftChild = Node(data)
2237                 return True
2238
2239         else:
2240             if self.rightChild:
2241                 return self.rightChild.insert(data)
2242             else:
2243                 self.rightChild = Node(data)
2244                 return True
2245
2246     def minValueNode(self, node):
2247         current = node
2248         while(current.leftChild is not None):
2249             current = current.leftChild
2250         return current
2251
2252     def delete(self, data):
2253         if self is None:
2254             return None
2255
2256         if data < self.data:
2257             self.leftChild = self.leftChild.delete(data)

```

```

2258         elif data > self.data:
2259             self.rightChild = self.rightChild.delete(data)
2260         else:
2261
2262             if self.leftChild is None:
2263                 temp = self.rightChild
2264                 self = None
2265                 return temp
2266             elif self.rightChild is None:
2267                 temp = self.leftChild
2268                 self = None
2269                 return temp
2270
2271             temp = self.minValueNode(self.rightChild)
2272             self.data = temp.data
2273             self.rightChild = self.rightChild.delete(temp.data)
2274         return self
2275
2276     def find(self, data):
2277         if(data == self.data):
2278             return True
2279         elif(data < self.data):
2280             if self.leftChild:
2281                 return self.leftChild.find(data)
2282             else:
2283                 return False
2284         else:
2285             if self.rightChild:
2286                 return self.rightChild.find(data)
2287             else:
2288                 return False
2289
2290     def preorder(self):
2291         if self:
2292             print(str(self.data), end=' ')
2293             if self.leftChild:
2294                 self.leftChild.preorder()
2295             if self.rightChild:
2296                 self.rightChild.preorder()
2297
2298     def inorder(self):
2299         if self:
2300             if self.leftChild:
2301                 self.leftChild.inorder()
2302             print(str(self.data), end=' ')
2303             if self.rightChild:
2304                 self.rightChild.inorder()
2305
2306     def postorder(self):
2307         if self:
2308             if self.leftChild:
2309                 self.leftChild.postorder()
2310             if self.rightChild:
2311                 self.rightChild.postorder()
2312             print(str(self.data), end=' ')
2313
2314
2315     class Tree(object):

```

```

2316     def __init__(self):
2317         self.root = None
2318
2319     def insert(self, data):
2320         if self.root:
2321             return self.root.insert(data)
2322         else:
2323             self.root = Node(data)
2324             return True
2325
2326     def delete(self, data):
2327         if self.root is not None:
2328             return self.root.delete(data)
2329
2330     def find(self, data):
2331         if self.root:
2332             return self.root.find(data)
2333         else:
2334             return False
2335
2336     def preorder(self):
2337         if self.root is not None:
2338             print()
2339             print('Preorder: ')
2340             self.root.preorder()
2341
2342     def inorder(self):
2343         print()
2344         if self.root is not None:
2345             print('Inorder: ')
2346             self.root.inorder()
2347
2348     def postorder(self):
2349         print()
2350         if self.root is not None:
2351             print('Postorder: ')
2352             self.root.postorder()
2353
2354
2355 if __name__ == '__main__':
2356     tree = Tree()
2357     tree.insert(10)
2358     tree.insert(12)
2359     tree.insert(5)
2360     tree.insert(4)
2361     tree.insert(20)
2362     tree.insert(8)
2363     tree.insert(7)
2364     tree.insert(15)
2365     tree.insert(13)
2366     print(tree.find(1))
2367     print(tree.find(12))
2368     '''
2369     Following tree is getting created:
2370           10
2371         /   \
2372        5     12
2373       /\       \

```

```

2374         4      8      20
2375         /      /
2376        7      15
2377         /
2378        13
2379
2380     tree.preorder()
2381     tree.inorder()
2382     tree.postorder()
2383     print('\n\nAfter deleting 20')
2384     tree.delete(20)
2385     tree.inorder()
2386     tree.preorder()
2387     print('\n\nAfter deleting 10')
2388     tree.delete(10)
2389     tree.inorder()
2390     tree.preorder()

```

```

2391
2392
2393 **LIST VIEW USING TREE**

```

```

2394
2395
2396 class Sample:
2397     def __init__(self, data_description, node_id, parent_id=""):
2398         self.data_description = data_description
2399         self.node_id = node_id
2400         self.parent_id = parent_id

```

```

2401
2402
2403 class Node:
2404     def __init__(self, data):
2405         self.data = Sample(data['data_description'],
2406                             data['node_id'], data['parent_id'])
2407         self.children = []

```

```

2408
2409
2410 class Tree:
2411     def __init__(self, data):
2412         self.Root = data
2413
2414     def insert_child(self, root, new_node):
2415         if root.data.node_id == new_node.data.parent_id:
2416             root.children.append(new_node)
2417
2418         elif len(root.children) > 0:
2419             for each_child in root.children:
2420                 self.insert_child(each_child, new_node)
2421
2422     def print_tree(self, root, point):
2423         print(point, root.data.node_id, root.data.parent_id,
2424               root.data.data_description)
2425         if len(root.children) > 0:
2426             point += "_"
2427             for each_child in root.children:
2428                 self.print_tree(each_child, point)

```

```

2429
2430

```

```
2431 data = {'data_description': 'Sample_root_1', 'node_id': '1', 'parent_id':  
2432 ''}  
2433 data1 = {'data_description': 'Sample_root_2', 'node_id': '2', 'parent_id':  
2434 '1'}  
2435 data2 = {'data_description': 'Sample_root_3', 'node_id': '3', 'parent_id':  
2436 '1'}  
2437 data3 = {'data_description': 'Sample_root_4', 'node_id': '4', 'parent_id':  
2438 '2'}  
2439 data4 = {'data_description': 'Sample_root_5', 'node_id': '5', 'parent_id':  
2440 '3'}  
2441 data5 = {'data_description': 'Sample_root_6', 'node_id': '6', 'parent_id':  
2442 '4'}  
2443 data6 = {'data_description': 'Sample_root_7', 'node_id': '7', 'parent_id':  
2444 '4'}  
2445  
2446 a = Tree(Node(data))  
2447 a.insert_child(a.Root, Node(data1))  
2448 a.insert_child(a.Root, Node(data2))  
2449 a.insert_child(a.Root, Node(data3))  
2450 a.insert_child(a.Root, Node(data4))  
2451 a.insert_child(a.Root, Node(data5))  
2452 a.insert_child(a.Root, Node(data6))  
2453 a.print_tree(a.Root, "|_")  
2454  
2455 **BREADTH FIRST TRAVERSAL**  
2456  
2457  
2458  
2459 class Node(object):  
2460     def __init__(self, data=None):  
2461         self.leftChild = None  
2462         self.rightChild = None  
2463         self.data = data  
2464  
2465     def height(node):  
2466         if node is None:  
2467             return 0  
2468         else:  
2469             leftHeight = height(node.leftChild)  
2470             rightHeight = height(node.rightChild)  
2471             if leftHeight > rightHeight:  
2472                 return leftHeight + 1  
2473             else:  
2474                 return rightHeight + 1  
2475  
2476     def breadthFirstTraversal(root):  
2477         if root == None:  
2478             return 0  
2479         else:  
2480             h = height(root)  
2481             for i in range(1, h + 1):  
2482                 printBFT(root, i)  
2483  
2484     def printBFT(root, level):  
2485         if root is None:
```

```

2482         return
2483     else:
2484         if level == 1:
2485             print(root.data, end=' ')
2486         elif level > 1:
2487             printBFT(root.leftChild, level - 1)
2488             printBFT(root.rightChild, level - 1)
2489
2490
2491 if __name__ == '__main__':
2492     root = Node(1)
2493     root.leftChild = Node(2)
2494     root.rightChild = Node(3)
2495     root.leftChild.leftChild = Node(4)
2496     breadthFirstTraversal(root)
2497
2498
2499 **COUNT LEAF NODES**
2500
2501
2502 def countLeafNodes(root):
2503     if root is None:
2504         return 0
2505     if (root.left is None and root.right is None):
2506         return 1
2507     else:
2508         return countLeafNodes(root.left) + countLeafNodes(root.right)
2509
2510
2511 if __name__ == '__main__':
2512     root = Node(1)
2513     root.setLeft(Node(2))
2514     root.setRight(Node(3))
2515     root.left.setLeft(Node(4))
2516     print('Count of leaf nodes:', countLeafNodes(root))
2517
2518
2519 **TREE FROM INORDER AND PREORDER**
2520
2521
2522 class Node:
2523     def __init__(self, data):
2524         self.data = data
2525         self.left = None
2526         self.right = None
2527
2528
2529 """Recursive function to construct binary of size len from
2530 Inorder traversal in[] and Preorder traversal pre[]. Initial values
2531 of start and end should be 0 and len -1. The function doesn't
2532 do any error checking for cases where inorder and preorder
2533 do not form a tree """
2534
2535
2536 def buildTree(inOrder, preOrder, start, end):
2537     if (start > end):
2538         return None
2539     tNode = Node(preOrder[buildTree.preIndex])

```

```

2540     buildTree.preIndex += 1
2541     if start == end:
2542         return tNode
2543     rootIndex = search(inOrder, start, end, tNode.data)
2544     tNode.left = buildTree(inOrder, preOrder, start, rootIndex-1)
2545     tNode.right = buildTree(inOrder, preOrder, rootIndex+1, end)
2546     return tNode
2547
2548
2549 def search(arr, start, end, value):
2550     for i in range(start, end+1):
2551         if arr[i] == value:
2552             return i
2553
2554
2555 def inorder(node):
2556     if node is None:
2557         return
2558     inorder(node.left)
2559     print(node.data, end=' ')
2560     inorder(node.right)
2561
2562
2563 inOrder = ['D', 'B', 'E', 'A', 'F', 'C']
2564 preOrder = ['A', 'B', 'D', 'E', 'C', 'F']
2565 buildTree.preIndex = 0
2566 root = buildTree(inOrder, preOrder, 0, len(inOrder)-1)
2567 print("Inorder traversal of the constructed tree is")
2568 inorder(root)
2569
2570
2571 **ROOT TO LEAF PATHS**
2572
2573
2574 class Node(object):
2575     def __init__(self, data=None):
2576         self.left = None
2577         self.right = None
2578         self.data = data
2579
2580
2581 def printPath(node, path=[]):
2582     if node is None:
2583         return
2584     path.append(node.data)
2585     if (node.left is None) and (node.right is None):
2586         print(' '.join([str(i) for i in path if i != 0]))
2587     else:
2588         printPath(node.left, path)
2589         printPath(node.right, path[0:1])
2590
2591
2592 if __name__ == '__main__':
2593     root = Node(1)
2594     root.left = Node(2)
2595     root.right = Node(3)
2596     root.left.left = Node(4)
2597     root.right.left = Node(5)

```

```
2598     printPath(root)
2599
2600
2601 **INORDER PREDECESSOR AND SUCCESSOR**
2602
2603
2604 class Node(object):
2605     def __init__(self, data):
2606         self.data = data
2607         self.left = None
2608         self.right = None
2609
2610     def findPredecessorAndSuccessor(self, data):
2611         global predecessor, successor
2612         predecessor = None
2613         successor = None
2614         if self is None:
2615             return
2616         if self.data == data:
2617             if self.left is not None:
2618                 temp = self.left
2619                 if temp.right is not None:
2620                     while(temp.right):
2621                         temp = temp.right
2622                 predecessor = temp
2623             if self.right is not None:
2624                 temp = self.right
2625                 while(temp.left):
2626                     temp = temp.left
2627                 successor = temp
2628             return
2629
2630         if data < self.data:
2631             print('Left')
2632             self.left.findPredecessorAndSuccessor(data)
2633         else:
2634             print('Right')
2635             self.right.findPredecessorAndSuccessor(data)
2636
2637     def insert(self, data):
2638         if self.data == data:
2639             return False
2640
2641         elif data < self.data:
2642             if self.left:
2643                 return self.left.insert(data)
2644             else:
2645                 self.left = Node(data)
2646                 return True
2647
2648         else:
2649             if self.right:
2650                 return self.right.insert(data)
2651             else:
2652                 self.right = Node(data)
2653                 return True
2654
2655
```



```
2656 if __name__ == '__main__':
2657     root = Node(50)
2658     root.insert(30)
2659     root.insert(20)
2660     root.insert(40)
2661     root.insert(70)
2662     root.insert(60)
2663     root.insert(80)
2664     root.findPredecessorAndSuccessor(70)
2665     if (predecessor is not None) and (successor is not None):
2666         print('Predecessor:', predecessor.data, 'Successor:',
successor.data)
2667     else:
2668         print('Predecessor:', predecessor, 'Successor:', successor)
2669
```