1. **Sort List**

#### Approach 1: Top Down Merge Sort

**Intuition**

Merge sort is a popularly known algorithm that follows the [Divide and Conquer Strategy](https://en.wikipedia.org/wiki/Divide-and-conquer_algorithm)(like quiecksort). The divide and conquer strategy can be split into 2 phases:

Divide phase: Divide the problem into subproblems.

Conquer phase: Repeatedly solve each subproblem independently and combine the result to form the original problem.

The Top Down approach for merge sort recursively splits the original list into sublists of equal sizes, sorts each sublist independently, and eventually merge the sorted lists. Let's look at the algorithm to implement merge sort in Top Down Fashion.

**Algorithm**

Recursively split the original list into two halves. The split continues until there is only one node in the linked list (Divide phase). To split the list into two halves, we find the middle of the linked list using the Fast and Slow pointer approach as mentioned in [Find Middle Of Linked List](https://leetcode.com/problems/middle-of-the-linked-list/).

class Solution:

def middleNode(self, head):

slow = fast = head

while fast and fast.next:

slow = slow.next

fast = fast.next.next

return slow

Recursively sort each sublist and combine it into a single sorted list. (Merge Phase). This is similar to the problem [Merge two sorted linked lists](https://leetcode.com/problems/merge-two-sorted-lists/)

def merge(self, h1, h2):

dummy = tail = ListNode(None)

while h1 and h2:

if h1.val < h2.val:

tail.next, tail, h1 = h1, h1, h1.next

else:

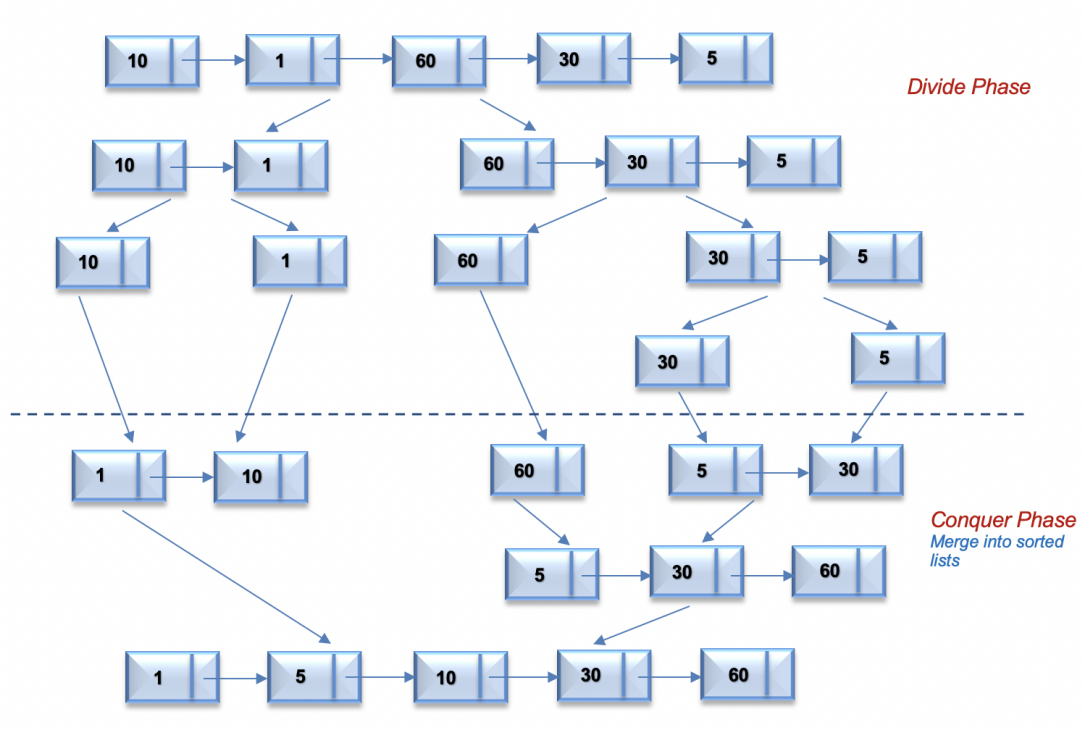
tail.next, tail, h2 = h2, h2, h2.next

tail.next = h1 or h2

return dummy.next

The process continues until we get the original list in sorted order.

For the linked list = [10,1,60,30,5], the following figure illustrates the merge sort process using a top down approach.



class Solution(object):

def merge(self, h1, h2):

dummy = tail = ListNode(None)

while h1 and h2:

if h1.val < h2.val:

tail.next, tail, h1 = h1, h1, h1.next

else:

tail.next, tail, h2 = h2, h2, h2.next

tail.next = h1 or h2

return dummy.next

def sortList(self, head):

if not head or not head.next:

return head

pre, slow, fast = None, head, head

while fast and fast.next:

pre, slow, fast = slow, slow.next, fast.next.next

pre.next = None

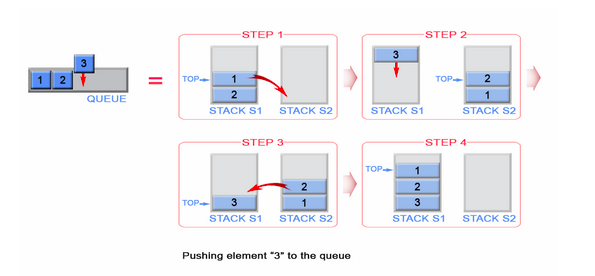
return self.merge(\*map(self.sortList, (head, slow)))

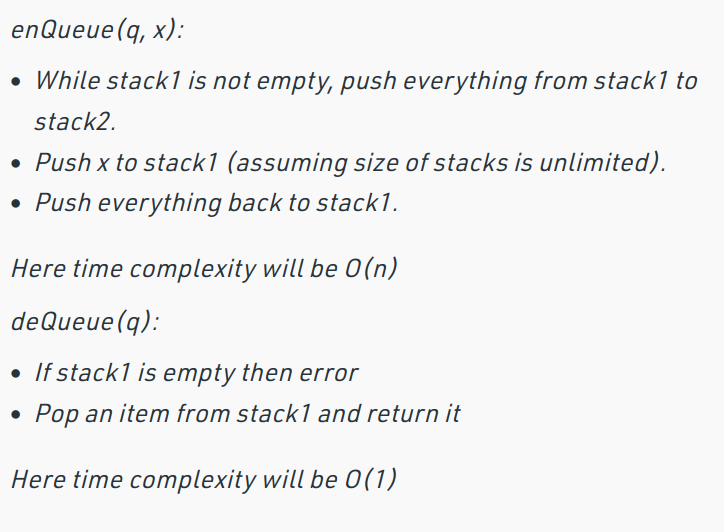
In conclusion:

We needed two capabilities to resolve the exercise

the ability to find the middle element in the list, and a second ability - to sort two sublist, in their scene, resolved the exercise

4. Implement Queue using Stacks





# Python3 program to implement Queue using

# two stacks with costly enQueue()

class Queue:

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

self.s1 = []

self.s2 = []

def enQueue(self, x):

# Move all elements from s1 to s2

while len(self.s1) != 0:

self.s2.append(self.s1[-1])

self.s1.pop()

# Push item into self.s1

self.s1.append(x)

# Push everything back to s1

while len(self.s2) != 0:

self.s1.append(self.s2[-1])

self.s2.pop()

# Dequeue an item from the queue

def deQueue(self):

# if first stack is empty

if len(self.s1) == 0:

print("Q is Empty")

# Return top of self.s1

x = self.s1[-1]

self.s1.pop()

return x

# Driver code

if \_\_name\_\_ == '\_\_main\_\_':

q = Queue()

q.enQueue(1)

q.enQueue(2)

q.enQueue(3)

print(q.deQueue())

print(q.deQueue())

print(q.deQueue())

# This code is contributed by PranchalK

-