```
# remove dups:

        current = head
        previous = None

        while current:
                if current.value in hash:
                        previous.next = current.next
                else:
                        hash.add(current.value)
                        previous = current
                current = current.next
        ll.tail = previous

        def remove_dup(self, ll):
                runner = current = ll
                while current:
                        runner = current
                        while runner.next:
                                if runner.next.value == current.value:
                                        runner.next = runner.next.next
                                else:
                                        runner = runner.next

                        current = current.next


# kth element to last
def k_th_element_to_last(self, ll):
        runner = current = ll.head()
        for _ in range(k):
                if not runner:
                        return None

                runner = runner.next

        while runner:
                current = current.next
                runner = runner.next




# partition
```

```
def partition (ll, x):
        current = ll.head
        current_next = current.next
        while current:
                if current.value < x:
                        current.next = ll.head
                        ll.head = current
                else:
                        ll.tail.next = current
                        ll.tail = current
                current = current_next

# reverse

def reverse(node):
        previous = None
        while node:
                next_node = node.next
                node.next = next_node.next
                next_node.next = node
                node = next_node


# for intersection

n1 = 0
while ll1:
        n1 += 1
        tail1 = ll1
        ll1 = n1.next
n2 = 0
while ll2:
        n2 += 1
        tail2 = ll2
        ll2 = n2.next

if tail1 ! = tail2:
        return False

p1=  ll1.head()
p2=  ll2.head()

if n1 > n2:
        bigger_ll = p1
```

```
        smaller_ll = p2
else:
        bigger_ll = p2
        smaller_ll = p1

for _ in range(abs(n1-n2)):
        bigger_ll = bigger_ll.next

while bigger_ll:
        if bigger_ll != smaller_ll:
                bigger_ll = bigger_ll.next
                smaller_ll = smaller_ll.next
        else:
                return bigger_ll

# loop/cycle

class Solution(object):
        def getIntersect(self, head):
                tortoise = head
                hare = head

                # A fast pointer will either loop around a cycle and meet the slow
                # pointer or reach the `null` at the end of a non-cyclic list.
                while hare is not None and hare.next is not None:
                        tortoise = tortoise.next
                        hare = hare.next.next
                        if tortoise == hare:
                                return tortoise

                return None

        def detectCycle(self, head):
                if head is None:
                        return None

                # If there is a cycle, the fast/slow pointers will intersect at some
                # node. Otherwise, there is no cycle, so we cannot find an entrance
        to
                # a cycle.
                intersect = self.getIntersect(head)
                if intersect is None:
                        return None

        # To find the entrance to the cycle, we have two pointers traverse at
```

```
        # the same speed -- one from the front of the list, and the other from
        # the point of intersection.
        ptr1 = head
        ptr2 = intersect
        while ptr1 != ptr2:
        ptr1 = ptr1.next
        ptr2 = ptr2.next

        return ptr1

class stack:
        def __init__(self, stack_size=10):
                self.stack_size = stack_size
                self.ele =[]

        def push(self, value):
                if len(self) < self.stack_size:
                        self.ele.append(value)
                else:
                        raise StackFullErrror()

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

        def pop(self):
                return self.ele.pop(value)

        def is_full(self):
                return len(self.ele) == stack_size

class stack_of_plates:
        def __init__(self, n_stacks, stack_size=100):
                self.n_stacks = n_stacks
                self.stack_size = stack_size
                self.stacks = [stack(), ]

        def push(self, value):
                if self.stacks[-1].is_full():
                        self.stacks.append(stack(self.stack_size))
                self.stacks[-1].push(value)

        def pop(self):
                val = self.stacks[-1].pop()
                if self.stacks[-1].is_empty():
                        self.stacks.pop()
```

```
def list_of_depths(root):
        bfs = deque()
        result = []
        current_level = -1
        bfs.add((root, 0))

        While bfs:
                node, depth = bfs.popleft()
                if depth == current_level:
                        result[-1].next = node
                else:
                        result.append(node)
                        current_level += 1

                if node.left: bfs.extend(node.left, depth+1)
                if node.right: bfs.extend(node.right, depth+1)



def depth (node):
        size = 1
        if node.left: size += depth(node.left)
        if node.right: size +=depth(node.right)
        return size

def check_balance(node):
        return abs(depth(node.left) - depth(node.right)) < 1
```

Populating next right pointers:

```
from collections import deque

def traverse(node, queue):
        if node1.left:
                queue.append(node1.left)
        if node2.right:
                queue.append(node1.right)
        return queue

def connect(self, root: 'Optional[Node]') -> 'Optional[Node]':
        queue = deque([root])
        current = deque([])
```

```
        while queue:
                n = len(queue)
                current = self.traverse(queue[0], current)
                for i1, i2 in zip(range(n), range(1, n)):
                        node1, node2 = queue[i1], queue[i2]
                        node1.next = node2
                        current = self.traverse(node2, current)

                        queue, current = current, []
            return root


def parser(k):
        string = reversed(s)
        ans = []
        cur_str = ''

        for i in string:
                if len(cur_str) == k:
                        ans.append(cur_str)
                        cur_str = ''
                elif i != '-':
                        if i.isalpha():
                                cur_str += i.upper()


        return '-'.join(reversed(ans))
```