DAA Assignment:-Linked List

1.Given the head of a singly linked list, reverse the list, and return *the reversed list*.

Sol:-

```
class Solution:
    def reverseList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        prev=None
        while head:
            prev,head.next,head=head,prev,head.next

        return prev
```

2. Given the head of a singly linked list, return *the middle node of the linked list*.

If there are two middle nodes, return **the second middle** node.

Sol:-

```
class Solution:
    def middleNode(self, head: Optional[ListNode]) -> Optional[ListNode]:
        temp=head
        c=0
        while temp:
            c+=1
            temp=temp.next
        temp=head
        for i in range(c//2):
            temp=temp.next

        return temp
```

3.Given the head of a linked list, remove the n^{th} node from the end of the list and return its head.

sol: -

```
def length(head):
    c=0
```

```
    if head==None:
        return c
    temp=head
    while temp:
        c+=1
        temp=temp.next
    return c

def removeKthNode(head, k):
    temp=head
    if head==None:
        return
    n=length(head)
    curr=head
    prev=None

    if n==k:
        return head.next

    for i in range(n-k):
        prev=curr
        curr=curr.next
    prev.next=curr.next
    curr.next=None

    return head
```

4. There is a singly-linked list head and we want to delete a node node in it.

Sol:-

```
        class Solution:
            def deleteNode(self, node):
                node.val=node.next.val
                node.next=node.next.next
```

5. Given the heads of two singly linked-lists headA and headB, return *the node at which the two lists intersect*. If the two linked lists have no intersection at all, return null.

Sol: -

```
def findIntersection(h1, h2):
    t1=h1
    t2=h2
    m={}
    while t1!=None:
        if(t1 not in m):
            m[t1]=1

        t1 = t1.next

    while(t2!=None):
        if(t2 in m):
            return t2

        t2 = t2.next

    return None
```

6. Given head, the head of a linked list, determine if the linked list has a cycle in it.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to. **Note that pos is not passed as a parameter**.

Return true *if there is a cycle in the linked list*. Otherwise, return false.

Sol: -

```
def detectCycle(head) :

    if head==None or head.next==None:

        return False
```

```
    slow=head
    fast=head.next
    while slow!=fast:
        if not fast or not fast.next:
            return False
        slow=slow.next
        fast=fast.next.next

    return True
```

7. Given the head of a singly linked list, return true *if it is a Palindrome or* false *otherwise*.

Sol: -

```
class Node:
    def __init__(self,data):

        self.data = data
        self.next = None


def isPalindrome(head):
    if head==None:
        return True
    temp=head
    l=[]
    while temp is not None:
        l.append(temp.data)
        temp=temp.next
    n=len(l)

    for i in range(len(l)):
        if l[i]!=l[n-i-1]:
            return False
    return True
```

8. Given the head of a linked list, return *the node where the cycle begins. If there is no cycle, return* null.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to (**0-indexed**). It is -1 if there is no cycle. **Note that** pos **is not passed as a parameter**.

Sol: -

```
def firstNode(head):
    slow=head
    fast=head
    while (slow and slow.next):
        fast=fast.next
        slow=slow.next.next
        if slow==fast:
            fast=head
            while fast!=slow:
                fast=fast.next
                slow=slow.next
            return slow
    return None
```