1. 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.

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def getIntersectionNode(self, headA: ListNode, headB: ListNode) -> ListNode:
        if not headA or not headB:
            return None

        # Two pointers
        pA, pB = headA, headB

        while pA != pB:
            # Move each pointer to the next node or switch to the other list's head
            pA = pA.next if pA else headB
            pB = pB.next if pB else headA

        return pA  # Either the intersection node or None
```

2. Delete a node in linked list

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None


class Solution:
    def deleteNode(self, node):
        """
        :type node: ListNode
        :rtype: void Do not return anything, modify node in-place instead.
        """
        node.val = node.next.val
        node.next = node.next.next
```

3. Delete a node from given position

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

def deleteNode(head, position):
    temp = head
    prev = None

    if temp is None:
        return head

    if position == 1:
        head = temp.next
        return head

    for i in range(1, position):
        prev = temp
        temp = temp.next
        if temp is None:
            print("Data not present")
            return head

    if temp is not None:
        prev.next = temp.next

    return head

def printList(head):
    while head:
        print(f"{head.data} -> ", end="")
        head = head.next
    print("None")

if __name__ == "__main__":
    head = Node(1)
    head.next = Node(2)
    head.next.next = Node(3)
    head.next.next.next = Node(4)
    head.next.next.next.next = Node(5)

    print("Original list: ", end="")
    printList(head)

    position = 2
    head = deleteNode(head, position)

    print("List after deletion: ", end="")
    printList(head)
```

4. 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.

```python
class SinglyLinkedListNode:
    def __init__(self, data):
        self.data = data
        self.next = None

def has_cycle(head):
    slow = head
    fast = head

    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

        if slow == fast:
            return True  # Cycle detected

    return False  # No cycle
```

5. Given the head of a singly linked list, reverse the list, and return the reversed list. (in C)

```c
#include <stdio.h>

struct Node {
   int data;
   struct Node* next;
};

struct Node* reverseList(struct Node* head) {
   struct Node *curr = head, *prev = NULL, *next;
   while (curr != NULL) {
      next = curr->next;
      curr->next = prev;
      prev = curr;
      curr = next;
   }
   return prev;
}

void printList(struct Node* node) {
   while (node != NULL) {
        printf(" %d", node->data);
        node = node->next;
    }
}

struct Node* createNode(int new_data) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = NULL;
    return new_node;
}

int main() {
    struct Node* head = createNode(1);
    head->next = createNode(2);
    head->next->next = createNode(3);
    head->next->next->next = createNode(4);
    head->next->next->next->next = createNode(5);

    printf("Given Linked list:");
    printList(head);

    head = reverseList(head);

    printf("\nReversed Linked List:");
    printList(head);

    return 0;
}
```

6. Swaps nodes in pair

```
class Solution:
    def swapPairs(self, head: Optional[ListNode]) -> Optional[ListNode]:
        if head is None or head.next is None:
            return head
        t = self.swapPairs(head.next.next)
        p = head.next
        p.next = head
        head.next = t
        return p
```

7. Rotate list by the kth place

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
```

```
class Solution:
    def rotateRight(self, head: Optional[ListNode], k: int) -> 
Optional[ListNode]:
        if head is None or head.next is None:
            return head
        cur, n = head, 0
        while cur:
            n += 1
            cur = cur.next
        k %= n
        if k == 0:
            return head
        fast = slow = head
        for _ in range(k):
            fast = fast.next
        while fast.next:
            fast, slow = fast.next, slow.next

        ans = slow.next
        slow.next = None
        fast.next = head
        return ans
```

8. Insert a node at given position

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

# function to insert a Node at required position
def insert_pos(head, pos, data):

    # This condition to check whether the
    # position given is valid or not.
    if pos < 1:
        return head

    # head will change if pos=1
    if pos == 1:
        new_node = Node(data)
        new_node.next = head
        size += 1
        return new_node

    curr = head

    # Traverse to the node that will be
    # present just before the new node
    for _ in range(1, pos - 1):
        if curr == None:
            break
        curr = curr.next

    # if position is greater
    # number of nodes
    if curr is None:
        return head

    new_node = Node(data)

    # update the next pointers
    new_node.next = curr.next
    curr.next = new_node

    return head

def print_list(head):
    curr = head
    while curr is not None:
        print(curr.data, end=" ")
        curr = curr.next
    print()

if __name__ == "__main__":

    # Creating the list 3->5->8->10
    head = Node(3)
    head.next = Node(5)
    head.next.next = Node(8)
    head.next.next.next = Node(10)

    data = 12
    pos = 3
    head = insert_pos(head, pos, data)
    print_list(head)
```

9. Given the head of a singly linked list, return true if it is a palindrome or false otherwise.

```
class Solution:
    def isPalindrome(self, head: Optional[ListNode]) -> bool:
        slow, fast = head, head.next
        while fast and fast.next:
            slow, fast = slow.next, fast.next.next
        pre, cur = None, slow.next
        while cur:
            t = cur.next
            cur.next = pre
            pre, cur = cur, t
        while pre:
            if pre.val != head.val:
                return False
            pre, head = pre.next, head.next
        return True
```

10. You are given the head of a linked list. Delete the middle node, and return the head of the modified linked list.
    The middle node of a linked list of size n is the ⌊n / 2⌋th node from the start using 0-based indexing, where ⌊x⌋ denotes the largest integer less than or equal to x.

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def deleteMiddle(self, head: Optional[ListNode]) -> Optional[ListNode]:
        dummy = ListNode(next=head)
        slow, fast = dummy, head
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next
        slow.next = slow.next.next
        return dummy.next
```

11. Merge two linked list

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
class Solution:
    def mergeTwoLists(
        self, list1: Optional[ListNode], list2: Optional[ListNode]
    ) -> Optional[ListNode]:
        if list1 is None or list2 is None:
            return list1 or list2
        if list1.val <= list2.val:
            list1.next = self.mergeTwoLists(list1.next, list2)
            return list1
        else:
            list2.next = self.mergeTwoLists(list1, list2.next)
            return list2
```