LINKED LISTS - SET 1 SOLUTIONS

Note: All solutions are written in C++.

Question 1:

- **Problem link**: https://practice.geeksforgeeks.org/problems/detect-loop-in-linked-list/1
- **Difficulty level**: Easy

Explanation:

Assign a new value to each data of node in the linked list which is not in the range given. Example suppose (1 <= Data on Node <= 10^3) then after visiting node assign the data as -1 as it is out of the given range. While traversing the linked list, if data of any node is -1, then we can conclude that there's a loop detected and return true.

Solution:

```
class Solution
{
    public:
    //Function to check if the linked list has a loop.
    bool detectLoop(Node* head)
    {
        if (!head)
            return 0;
        else {
            while (head) {
                if (head->data == -1) {
                    return true;
                }
                else {
                    head->data = -1;
```

```
                    head = head->next;
                }
            }
            return 0;
        }
    }
};
```

Complexity:

- Time: O(n)
- Space: O(1)

Question 2:

- **Problem link**: https://leetcode.com/problems/reverse-linked-list/
- **Difficulty level**: Easy

Explanation:

Maintain three pointers which will point to the previous, current and next nodes in the list. In every step make the current node point to the previous and shift all the three pointers by one unit. Do this until the end of the list and finally, the previous will point to the last element which will be the first element in the reverse list.

Solution:

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode *curr,*nextnode,*prev;
        if(!head) return head; //empty list
        curr=head;
        prev=NULL;
```

```
        nextnode=curr->next;

        while(curr)
        {
            curr->next=prev;
            prev=curr;
            curr=nextnode;
            if(nextnode) nextnode=curr->next;
        }

        return prev;
    }
};
```

Complexity:

- Time: O(n)
- Extra space: O(1)

Question 3:

- **Problem link**: https://practice.geeksforgeeks.org/problems/remove-duplicate-element-from-sorted-linked-list/1
- **Difficulty level**: Easy

Explanation:

Create a pointer that will point towards the first occurrence of every element and another pointer temp which will iterate to every element and when the value of the previous pointer is not equal to the temp pointer, we will set the pointer of the previous pointer to the first occurrence of another node.

Solution:

```
void removeDuplicates(struct Node* head)
{
    struct Node* temp = head;
    struct Node* prev=head;

    while (temp != NULL)
    {

        if(temp->data != prev->data)
        {
            prev->next = temp;
            prev = temp;
        }

        temp = temp->next;
    }
    if(prev != temp)
    {
        prev->next = NULL;
    }

}
```

Complexity:

- Time: O(n)
- Space: O(1)

Question 4:

- **Problem link**: https://leetcode.com/problems/intersection-of-two-linked-lists/
- **Difficulty level**: Easy

Explanation:

We can use two iterations to do that. In the first iteration, we will reset the pointer of one linkedlist to the head of another linkedlist after it reaches the tail node. In the second iteration, we will move two pointers until they point to the same node. Our operations in the first iteration will help us counteract the difference. So if two linked lists intersect, the meeting point in the second iteration must be the intersection point. If the two linked lists have no intersection at all, then the meeting pointer in second iteration must be the tail node of both lists, which is null

Solution:

```
ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
    ListNode *ptrA = headA, *ptrB = headB;
    while (ptrA != ptrB) {
        ptrA = ptrA ? ptrA->next : headB;
        ptrB = ptrB ? ptrB->next : headA;
    }
    return ptrA;
}
```

Complexity:

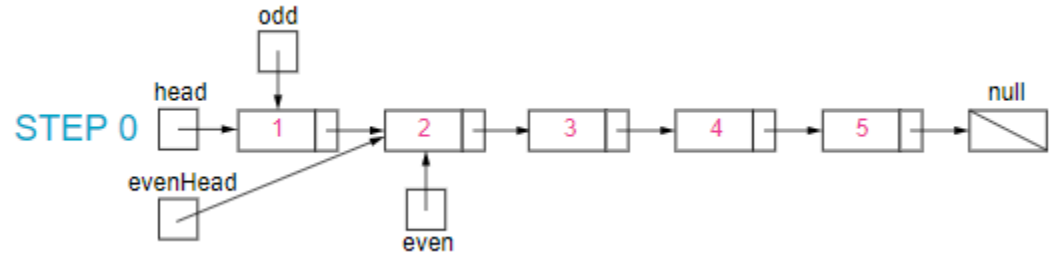
- Time: O(n)
- Space: O(1)

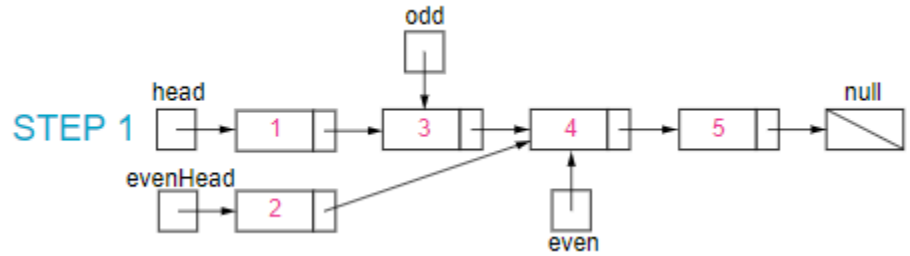
Question 5:

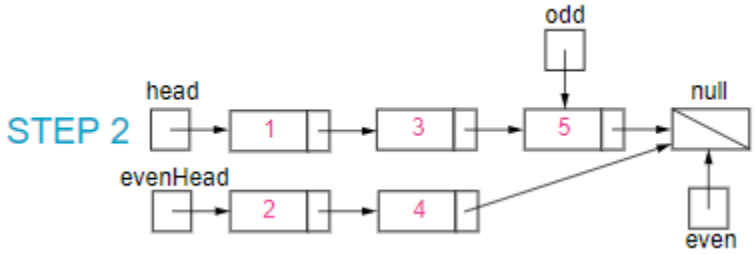
- **Problem link**: https://leetcode.com/problems/odd-even-linked-list/
- **Difficulty level**: Medium

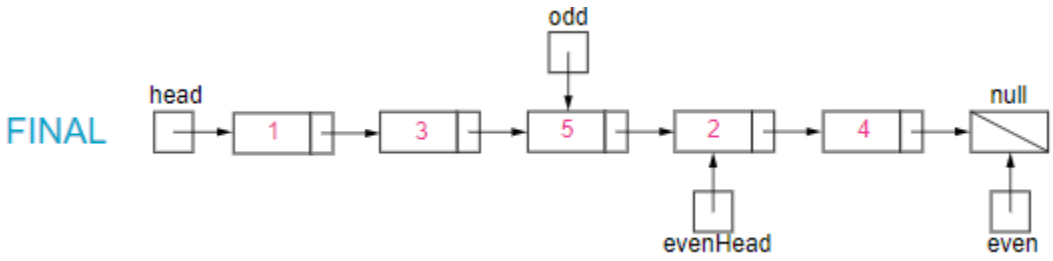
Explanation:

Put the odd nodes in a linked list and the even nodes in another. Then link the evenList to the tail of the oddList. An illustration of our algorithm is following:









Solution:

```
ListNode* oddEvenList(ListNode* head)
 {
     if(!head) return head;
     ListNode *odd=head, *evenhead=head->next, *even = evenhead;
     while(even && even->next)
     {
         odd->next = odd->next->next;
         even->next = even->next->next;
         odd = odd->next;
         even = even->next;
     }
     odd->next = evenhead;
     return head;
 }
```

Complexity:

- Time: O(n)
- Space: O(1)

Question 6:

- **Problem link**: https://leetcode.com/problems/rotate-list/
- **Difficulty level**: Medium

Explanation: first find out the length of the linkedlist. Then make it circular by referencing the last node next pointer to head. Now break the linkedlist after the [n-k%n] node to make it non-circular again. Return pointer next to [n-k%n] node.

Solution:

```
class Solution {
public:
    ListNode* rotateRight(ListNode* head, int k) {
        if(head==NULL) return head;

        int n=1;
        ListNode* temp=head;

        while(temp->next){
            n++;
            temp=temp->next;
        }
        temp->next=head;

        temp=head;
        for(int i=1; i<n-k%n; i++)
            temp=temp->next;
        head=temp->next;
        temp->next=NULL;

        return head;
    }
};
```

Complexity:

- Time: O(n)
- Space: O(1)

Question 7:

- **Problem link**: https://leetcode.com/problems/add-two-numbers/
- **Difficulty level**: Medium

Explanation: refer to the solution tab of the problem for a better explanation.
https://leetcode.com/problems/add-two-numbers/solution/

Solution:

```
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        ListNode dummy(0);
        ListNode* curr=& dummy;

        int carry=0 , x , y , sum;

        while(l1 or l2){
            x=l1?l1->val:0;
            y=l2?l2->val:0;

            sum=x+y+carry;
            carry=sum/10;

            curr->next=new ListNode(sum%10);
            curr=curr->next;

            if(l1) l1=l1->next;
            if(l2) l2=l2->next;
        }

        if(carry==1)
            curr->next=new ListNode(carry);
```

```
        return dummy.next;
    }
};
```

Complexity:

- Time: O(max(m,n))
- Space: O(max(m,n))

Question 8:

- **Problem link**: https://practice.geeksforgeeks.org/problems/clone-a-linked-list-with-next-and-random-pointer/1
- **Difficulty level**: Hard

Explanation:

- Create the copy of node 1 and insert it between node 1 & node 2 in the original Linked List, create a copy of 2 and insert it between 2 & 3. Continue in this fashion, add the copy of N after the Nth node.
- Now copy the random link in this fashion
  original->next->random= original->random->next;
  /*TRAVERSE TWO NODES*/
- This works because original->next is nothing but a copy of the original and Original->random->next is nothing but a copy of the random.

- Now restore the original and copy linked lists in this fashion in a single loop.
  original->next->random= original->random->next;
  /*TRAVERSE TWO NODES*/
- Ensure that original->next is NULL and return the cloned list.

Solution:

```
Node *copyList(Node *head) {
    Node* curr = head, *temp;

    // insert additional node after every node of original list
    while (curr)
    {
        temp = curr->next;

        // Inserting node
        curr->next = new Node(curr->data);
        curr->next->next = temp;
        curr = temp;
    }

    curr = head;

    // adjust the random pointers of the newly added nodes
    while (curr)
    {
        if(curr->next)
            curr->next->arb = curr->arb ? curr->arb->next : curr->arb;

        // move to the next newly added node by skipping an original node
        curr = curr->next?curr->next->next:curr->next;
    }

    Node* original = head, *copy = head->next;

    // save the start of copied linked list
    temp = copy;

    // now separate the original list and copied list
    while (original && copy)
    {
```

```
        original->next =
         original->next? original->next->next : original->next;

        copy->next = copy->next?copy->next->next:copy->next;
        original = original->next;
        copy = copy->next;
    }

    return temp;
}
```

Complexity:

- Time: O(n)
- Space: O(1)

Question 9:

- **Problem link**: https://leetcode.com/problems/reverse-nodes-in-k-group/
- **Difficulty level**: Hard

Explanation:

To reverse every group, use the same logic of maintaining three pointers namely 'prev' , 'curr' and 'nxt' as done in Question 2. Extending the same logic, use two more pointers 'start' and 'bef_start'. The pointer 'start' will point to the first element of every group **before reversing** and 'bef_start' will point to the last element of the previous group **after reversing** the group.

As the number of groups will be '**n/k'**, where n is the number of elements in the original list, run the outer loop 'n/k' times reversing a group of 'k' elements in each iteration.
Solution using another approach can be found here.

Solution:

```
class Solution {
public:
    ListNode* reverseKGroup(ListNode* head, int k) {
        ListNode *start,*curr,*prev,*nxt,*bef_start;

        //count number of nodes in list
        int n=0;
        curr=head;
        while(curr)
        {
            n++;
            curr=curr->next;
        }

        //initialize the pointers
        curr = head;
        prev = NULL;
        nxt = curr->next;
        int num_times = n/k;  //number of groups

        for(int i=0;i<num_times;i++)
        {
            int temp = k; //number of nodes to be reversed
            bef_start = start;// points to last element of previous group
            start = curr; //assign start as the first element in non-reversed group
            while(temp--) //same as in Q2
            {
                curr->next=prev;
```

```
                prev=curr;
                curr=nxt;
                if(nxt) nxt=nxt->next;

            }
            if(i==0) // update head in first iteration
            {
                head=prev;
            }
            else // update the 'next' of last element of the previous group
            {
                bef_start->next=prev;
            }

            //update prev as group is now reversed
            prev=start;

            //update as start has become the
            //last element in the group after reversing
            start->next=curr;
        }

        return head;
    }
};
```

Complexity:

- Time: O(n)
  Proof: As the number of groups will be '**n/k**' , to reverse every group of '**k**' elements it will take O(k) time. Hence **(n/k)*k=n** giving us O(n) time.
- Space: O(1)