Assignment 14 (Linked List) Solution

Question 1

Given a linked list of **N** nodes such that it may contain a loop.

A loop here means that the last node of the link list is connected to the node at position X(1-based index). If the link list does not have any loop, X=0.

Remove the loop from the linked list, if it is present, i.e. unlink the last node which is forming the loop.

Solution

```
int detectAndRemoveLoop(struct Node* list)
{
    struct Node *slow_p = list, *fast_p = list;

    // Iterate and find if loop exists or not
    while (slow_p && fast_p && fast_p->next) {
        slow_p = slow_p->next;
        fast_p = fast_p->next->next;

        /* If slow_p and fast_p meet at some point then there
           is a loop */
        if (slow_p == fast_p) {
            removeLoop(slow_p, list);
```

```
            /* Return 1 to indicate that loop is found */
            return 1;
        }
    }

    /* Return 0 to indicate that there is no loop*/
    return 0;
}

/* Function to remove loop.
 loop_node --> Pointer to one of the loop nodes
 head -->  Pointer to the start node of the linked list */
void removeLoop(struct Node* loop_node, struct Node* head)
{
    struct Node* ptr1 = loop_node;
    struct Node* ptr2 = loop_node;

    // Count the number of nodes in loop
    unsigned int k = 1, i;
    while (ptr1->next != ptr2) {
        ptr1 = ptr1->next;
        k++;
```

```
}

// Fix one pointer to head
ptr1 = head;

// And the other pointer to k nodes after head
ptr2 = head;
for (i = 0; i < k; i++)
    ptr2 = ptr2->next;

/*  Move both pointers at the same pace,
  they will meet at loop starting node */
while (ptr2 != ptr1) {
    ptr1 = ptr1->next;
    ptr2 = ptr2->next;
}

// Get pointer to the last node
while (ptr2->next != ptr1)
    ptr2 = ptr2->next;

/* Set the next node of the loop ending node
  to fix the loop */
```

```
    ptr2->next = NULL;
}
```

Question 2

A number N is represented in Linked List such that each digit corresponds to a node in linked list. You need to add 1 to it.

Solution:

```
Node* reversse(Node *head)
   {


      if(head==NULL){
         return head;
      }


      Node * prev=NULL;
      while(head!=NULL){
         Node * next=head->next;
         head->next=prev;
         prev=head;
         head=next;
      }



      return prev;
      // code here
```

```
    // return head of reversed list
}
Node* addOne(Node *head)
{
    Node* ll=reversse(head);

    int sum=ll->data+1;
    int carry=sum/10;
    ll->data=sum%10;

    Node*temp=NULL;
    Node* ans=ll;
    ll=ll->next;

    while(ll!=NULL){
        sum=carry+ll->data;

        if(sum>=10) {
            carry=1;
        }
        else carry=0;

        sum=sum%10;

        ll->data=sum;
```

```
        ll=ll->next;
    }

    if(carry>0){
        head->next=new Node(carry);
    }

    return reversse(ans);
  }
```

Question 3

Given a Linked List of size N, where every node represents a sub-linked-list and contains two pointers:(i) a **next** pointer to the next node,(ii) a **bottom** pointer to a linked list where this node is head.Each of the sub-linked-list is in sorted order.Flatten the Link List such that all the nodes appear in a single level while maintaining the sorted order. **Note:** The flattened list will be printed using the bottom pointer instead of next pointer.

Solution:

```
public Node merge(Node l1,Node l2){
    Node dummy=new Node(-1);
    Node nhead=dummy;
    while(l1!=null&&l2!=null){
        if(l1.data<l2.data){
          Node n=new Node(l1.data);
          dummy.bottom=n;
          dummy=n;
```

```
            l1=l1.bottom;
          }else{
            Node n=new Node(l2.data);
            dummy.bottom=n;
            dummy=n;
            l2=l2.bottom;
          }
        }
        while(l1!=null){
            Node n=new Node(l1.data);
            dummy.bottom=n;
            dummy=n;
            l1=l1.bottom;
        }
        while(l2!=null){
           Node n=new Node(l2.data);
            dummy.bottom=n;
            dummy=n;
            l2=l2.bottom;
        }
        return nhead.bottom;



}
Node flatten(Node root)
```

```
{
        // Your code here
        if(root==null||root.next==null){
            return root;
        }
        root.next=flatten(root.next);
        root=merge(root,root.next);
        return root;



}
```

Question 4

You are given a special linked list with **N** nodes where each node has a next pointer pointing to its next node. You are also given **M** random pointers, where you will be given **M** number of pairs denoting two nodes **a** and **b i.e. a->arb = b** (arb is pointer to random node)**.**

Construct a copy of the given list. The copy should consist of exactly **N** new nodes, where each new node has its value set to the value of its corresponding original node. Both the next and random pointer of the new nodes should point to new nodes in the copied list such that the pointers in the original list and copied list represent the same list state. None of the pointers in the new list should point to nodes in the original list.

For example, if there are two nodes **X** and **Y** in the original list, where **X.arb --> Y**, then for the corresponding two nodes **x** and **y** in the copied list, **x.arb --> y.**

Return the head of the copied linked list.

Solution:

```
public RandomListNode copyRandomList(RandomListNode head) {

  RandomListNode iter = head, next;
```

```
// First round: make copy of each node,
// and link them together side-by-side in a single list.
while (iter != null) {
  next = iter.next;

  RandomListNode copy = new RandomListNode(iter.label);
  iter.next = copy;
  copy.next = next;

  iter = next;
}

// Second round: assign random pointers for the copy nodes.
iter = head;
while (iter != null) {
  if (iter.random != null) {
    iter.next.random = iter.random.next;
  }
  iter = iter.next.next;
}

// Third round: restore the original list, and extract the copy list.
iter = head;
RandomListNode pseudoHead = new RandomListNode(0);
RandomListNode copy, copyIter = pseudoHead;
```

```
  while (iter != null) {
    next = iter.next.next;

    // extract the copy
    copy = iter.next;
    copyIter.next = copy;
    copyIter = copy;

    // restore the original list
    iter.next = next;

    iter = next;
  }

  return pseudoHead.next;
}
```

Question 5

Given the `head` of a singly linked list, group all the nodes with odd indices together followed by the nodes with even indices, and return *the reordered list*.

The **first** node is considered **odd**, and the **second** node is **even**, and so on.

Note that the relative order inside both the even and odd groups should remain as it was in the input.

You must solve the problem in `O(1)` extra space complexity and `O(n)` time complexity.

Solution:

```
static Node rearrangeEvenOdd(Node head)
{
    // Corner case
    if (head == null)
        return null;

    // Initialize first nodes of even and
    // odd lists
    Node odd = head;
    Node even = head.next;

    // Remember the first node of even list so
    // that we can connect the even list at the
    // end of odd list.
    Node evenFirst = even;

    while (1 == 1)
    {
        // If there are no more nodes,
        // then connect first node of even
        // list to the last node of odd list
        if (odd == null || even == null ||
                    (even.next) == null)
        {
            odd.next = evenFirst;
```

```
            break;
        }

        // Connecting odd nodes
        odd.next = even.next;
        odd = even.next;

        // If there are NO more even nodes
        // after current odd.
        if (odd.next == null)
        {
            even.next = null;
            odd.next = evenFirst;
            break;
        }

        // Connecting even nodes
        even.next = odd.next;
        even = odd.next;
    }
    return head;
}
```

Question 6

Given a singly linked list of size **N**. The task is to **left-shift** the linked list by **k** nodes, where **k** is a given positive integer smaller than or equal to length of the linked list.

Solution:

```
 void rotate(int k)
   {
       if (k == 0)
           return;

       // Let us understand the below code for example k =
       // 4 and list = 10->20->30->40->50->60.
       Node current = head;

       // current will either point to kth or NULL after
       // this loop. current will point to node 40 in the
       // above example
       int count = 1;
       while (count < k && current != null) {
           current = current.next;
           count++;
       }

       // If current is NULL, k is greater than or equal to
       // count of nodes in linked list. Don't change the
       // list in this case
       if (current == null)
           return;
```

```
        // current points to kth node. Store it in a
        // variable. kthNode points to node 40 in the above
        // example
        Node kthNode = current;

        // current will point to last node after this loop
        // current will point to node 60 in the above
        // example
        while (current.next != null)
            current = current.next;

        // Change next of last node to previous head
        // Next of 60 is now changed to node 10

        current.next = head;

        // Change head to (k+1)th node
        // head is now changed to node 50
        head = kthNode.next;

        // change next of kth node to null
        kthNode.next = null;
    }
```

Question 7

You are given the `head` of a linked list with `n` nodes.

For each node in the list, find the value of the **next greater node**. That is, for each node, find the value of the first node that is next to it and has a **strictly larger** value than it.

Return an integer array `answer` where `answer[i]` is the value of the next greater node of the `ith` node (**1-indexed**). If the `ith` node does not have a next greater node, set `answer[i] = 0`.

Solution:

```
 public int[] nextLargerNodes(ListNode head) {
        ListNode prev = null;
        int n = 0;
        for (ListNode node = head, next; node != null; node = next) {
            next = node.next;
            node.next = prev;
            prev = node;
            n++;
        }
        int[] ans = new int[n], stack = new int[n];
        int top = 0;
        ListNode node = prev;
        for (int i = n - 1; i >= 0; i--) {
            while (top > 0 && stack[top - 1] <= node.val) {
                top--;
            }
            ans[i] = top > 0 ? stack[top - 1] : 0;
            stack[top++] = node.val;
            node = node.next;
        }
```

```
    return ans;
}
```

Question 8

Given the `head` of a linked list, we repeatedly delete consecutive sequences of nodes that sum to `0` until there are no such sequences.

After doing so, return the head of the final linked list. You may return any such answer.

(Note that in the examples below, all sequences are serializations of `ListNode` objects.)

Solution:

```
public ListNode removeZeroSumSublists(ListNode head) {
    int sum = 0;
ListNode dm = new ListNode(0);
dm.next = head;

Map<Integer, ListNode> mp = new HashMap<>();
mp.put(0, dm);


for (ListNode i = dm; i != null; i = i.next) {
    sum += i.val;
    mp.put(sum, i);
}


sum = 0;
```

```
for (ListNode i = dm; i != null; i = i.next) {
    sum += i.val;
    i.next = mp.get( sum ).next;
}

return dm.next;
}
```