# LINKED LISTS - SET 3 SOLUTIONS

Note: All solutions are written in C++.

## Question 1:

- **Problem link**: https://practice.geeksforgeeks.org/problems/count-pairs-whose-sum-is-equal-to-x/1
- **Difficulty level**: Easy

**Explanation:**

We'll use the concept of hashing. Hash tables are implemented using unordered_set in C++. We store all first linked list elements in a hash table. For elements of the second linked list, we subtract every element from **x** and check the result in the hash table. If the result is present, we increment the **count**.

**Solution:**

```
class Solution{
  public:
    int countPairs(struct Node* head1, struct Node* head2, int x) {
        int count = 0;

        unordered_set<int> us;

        // insert all the elements of 1st list in the hash
        // table(unordered_set 'us')
        while (head1 != NULL)
        {
            us.insert(head1->data);
```

```
            // move to next node
            head1 = head1->next;
        }

        // for each element of 2nd list
        while (head2 != NULL)
        {
            // find (x - head2->data) in 'us'
            if (us.find(x - head2->data) != us.end())
                count++;

            // move to next node
            head2 = head2->next;
        }
        // required count of pairs
        return count;
    }
};
```

**Complexity:**

- Time: O(M+N)
- Space: O(M+N)

## Question 2:

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

**Explanation:**

Use fast and slow pointers : When traversing the list with a pointer `slow`, make another pointer `fast` that traverses twice as fast. When `fast` reaches the end of the list, `slow` must be in the middle.

**Solution:**

```
class Solution {
public:
    ListNode* middleNode(ListNode* head) {
        ListNode* slow = head;
        ListNode* fast = head;
        while (fast != NULL && fast->next != NULL) {
            slow = slow->next;
            fast = fast->next->next;
        }
        return slow;
    }
};
```

**Complexity:**

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

## Question 3:

- **Problem link**: https://practice.geeksforgeeks.org/problems/add-1-to-a-number-represented-as-linked-list/1
- **Difficulty level**: Easy

**Explanation:**

The main focus in this question is on the digit 9 which creates all the changes otherwise for other digits we have to just increment their value by 1 but if we change the node's value with the value 9 it makes a carry which then has to be passed through the linked list.

Find the last node in the linked list which is not equal to 9. Now there are three cases:

1. If there is no such node i.e. the value of every node is 9 then the new linked list will contain all 0s and a single 1 inserted at the head of the linked list.
2. If the rightmost node which is not equal to 9 is the last node in the linked list then add 1 to this node and return the head of the linked list.
3. If the node is other than the last node i.e. every node after it is equal to 9 then add 1 to the current node and change all the nodes after it to 0.

**Solution:**

```
Node* addOne(Node* head){
    Node* last = NULL;
    Node* cur = head;
    while (cur->next != NULL) {
        if (cur->data != 9) {
            last = cur;
        }
        cur = cur->next;
    }
    if (cur->data != 9) {
        cur->data++;
        return head;
    }
    if (last == NULL) {
        last = new Node();
        last->data = 0;
        last->next = head;
        head = last;
    }
    last->data++;
    last = last->next;
    while (last != NULL) {
        last->data = 0;
        last = last->next;
    }
    return head;
```

```
}
```

**Complexity:**

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

## Question 4:

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

**Explanation:**

We take two pointers 'before' and 'after' to keep track of the two linked lists, the first pointing to a list with elements less than 'x' and the second pointing to the list with elements greater than or equal to 'x'. Iterating through the original list, if an element is smaller than 'x' then move the node to 'before' otherwise move it to list with the pointer 'after'. Finally, combine the two lists.

**Solution:**

```
class Solution {
public:
    ListNode* partition(ListNode* head, int x) {

        ListNode *before_head = new ListNode(0);
        ListNode *before = before_head;
        ListNode *after_head = new ListNode(0);
        ListNode *after = after_head;

        while (head != NULL) {
```

```
            if (head->val < x) {
                before->next = head;
                before = before->next;
            } else {
                after->next = head;
                after = after->next;
            }

            head = head->next;
        }

        after->next = NULL;
        before->next = after_head->next;

        return before_head->next;
    }
};
```

**Complexity:**

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

## Question 5:

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

**Explanation:**

Use two pointers, one of them taking one step at a time and another takes two steps. Suppose the first meet at step 'k' , the length of the cycle is 'r'. So, 2k-k=nr, k=nr. Now, the distance between the start node of the list and the start node of the cycle is 's'. The distance between the start of the list and the first meeting node is k(the pointer which walks one step at a time has walked k steps). The distance between the start node of cycle and the first meeting node is 'm', so...s=k-m, s=nr-m=(n-1)r+(r-m),here we take n = 1, So, using one pointer start from the head of list, another pointer start from the first meeting node, all of them walk one step at a time, the first time they meet each other is the start of the cycle.

A better diagramatic explanation of the proof and solution can be found in Approach 3 in the link.

**Solution:**

```
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        if (head == NULL || head->next == NULL) return NULL;

        ListNode* firstp = head;
        ListNode* secondp = head;
        bool isCycle = false;

        while(firstp != NULL && secondp != NULL) {
            firstp = firstp->next;
            if (secondp->next == NULL) return NULL;
            secondp = secondp->next->next;
            if (firstp == secondp) { isCycle = true; break; }
        }

        if(!isCycle) return NULL;
        firstp = head;
```

```
        while( firstp != secondp) {
            firstp = firstp->next;
            secondp = secondp->next;
        }

        return firstp;
    }
};
```

**Complexity:**

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

## Question 6:

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

**Explanation:**

Traverse both the list and store the data into two new vectors. You can use stack instead of vector to store data value. Now Sum both the vectors from lower unit place to higher and keep calculating carry until all the elements are accessed. At last if carry==1 then insert a new node with value 1.

**Solution:**

```
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        int n1=0 , n2=0;
        vector<int> nums1 , nums2;
        ListNode *list1=l1 , *list2=l2;

        while(list1){
```

```
            n1++;
            nums1.push_back(list1->val);
            list1=list1->next;
        }
        while(list2){
            n2++;
            nums2.push_back(list2->val);
            list2=list2->next;
        }

        int sum;
        int carry=0;
        ListNode* pre=NULL;
        while(n1>0 or n2>0){
            sum=carry;
            if(n1>0)
                sum+=nums1[--n1];
            if(n2>0)
                sum+=nums2[--n2];

            ListNode* temp = new ListNode(sum%10);
            carry=sum/10;
            temp->next=pre;
            pre=temp;
        }
        if(carry){
            ListNode* temp = new ListNode(carry);
            temp->next=pre;
            pre=temp;
        }
        return pre;
    }
};
```

**Complexity:**

- Time: O(n+m)
- Space: O(n+m)

## Question 7:

- **Problem link**: https://leetcode.com/problems/split-linked-list-in-parts/
- **Difficulty level**: Medium

**Explanation:**

First calculate the length of the list. each part has size `L = width + (i < remainder ? 1 : 0)`, . We create a new list and write the part to that list.

For a more clear picture and explanation refer here.

**Solution:**

```
class Solution {
public:
    vector<ListNode*> splitListToParts(ListNode* root, int k) {
        ListNode *temp=root;
        int n=0;

        // count number of nodes present in the list
        while(temp){
            n++;
            temp=temp->next;
        }

        int width=n/k , rem=n%k;

        vector<ListNode*> ans(k,NULL);

        temp=root;
        // for loop for each part
```

```
        for(int i=0; i<k; i++){
            ListNode *head = temp;
            // for loop for length of each part
            for(int j=0; j<width+(i<rem?1:0)-1 ; j++){
                if(temp)
                    temp = temp->next;
            }
            // put null to last node->next pointer of each part
            if(temp){
                ListNode *prev=temp;
                temp = temp->next;
                prev->next=NULL;
            }
            // store the part
            ans[i]=head;
        }
        return ans;
    }
};
```

**Complexity:**

- Time: O(n+k)
- Space: O(k)

## Question 8:

- **Problem link**: https://practice.geeksforgeeks.org/problems/length-of-longest-palindrome-in-linked-list/1
- **Difficulty level**: Hard

**Explanation:**

The idea is based on iterative linked list reverse process. We iterate through the given linked list and one by one reverse every prefix of the linked list from the left. After reversing a prefix, we find the longest common list beginning from reversed prefix and the list after the reversed prefix.

**Solution:**

```
int countCommon(Node *a, Node *b)
{
    int count = 0;

    // loop to count common in the list starting
    // from node a and b
    for (; a && b; a = a->next, b = b->next)

        // increment the count for same values
        if (a->data == b->data)
            ++count;
        else
            break;

    return count;
}

/*The function below returns an int denoting
the length of the longest palindrome list. */

int maxPalindrome(Node *head)
{
    //Your code here
    int result = 0;
    Node *prev = NULL, *curr = head;

    // loop till the end of the linked list
    while (curr)
```

```
    {
        // The sublist from head to current
        // reversed.
        Node *next = curr->next;
        curr->next = prev;

        // check for odd length palindrome
        // by finding longest common list elements
        // beginning from prev and from next (We
        // exclude curr)
        result = max(result,
                     2*countCommon(prev, next)+1);

        // check for even length palindrome
        // by finding longest common list elements
        // beginning from curr and from next
        result = max(result,
                     2*countCommon(curr, next));

        // update prev and curr for next iteration
        prev = curr;
        curr = next;
    }
    return result;
}
```

**Complexity:**

- Time: $O(n^2)$
- Space: $O(1)$

## Question 9:

- **Problem link**: https://practice.geeksforgeeks.org/problems/sort-a-linked-list/1
- **Difficulty level**: Hard

**Explanation:**

**mergeSort():**

1. If the size of the linked list is 1 then return the head
2. Find mid using logic of Q2
3. Store the next of mid in head2 i.e. the right sub-linked list.
4. Now Make the next midpoint null.
5. Recursively call mergeSort() on both left and right sub-linked lists and store the new head of the left and right linked list.
6. Call merge() given the arguments of new heads of left and right sub-linked lists and store the final head returned after merging.
7. Return the final head of the merged linked list.

**merge(head1, head2):**

1. Take a pointer say merged to store the merged list in it and store a dummy node in it.
2. Take a pointer temp and assign merge to it.
3. If the data of head1 is less than the data of head2, then, store head1 in next of temp & move head1 to the next of head1.
4. Else store head2 in next of temp & move head2 to the next of head2.
5. Move temp to the next of temp.
6. Repeat steps 3, 4 & 5 until head1 is not equal to null and head2 is not equal to null.
7. Now add any remaining nodes of the first or the second linked list to the merged linked list.
8. Return the next of merged(that will ignore the dummy and return the head of the final merged linked list)

**Solution:**

```
Node* mergeSort(Node* head){
      if (head == NULL || head->next == NULL)
            return head;
      Node* slowPtr = head, *fastPtr = head, *prev = NULL;
      while (fastPtr != NULL && fastPtr->next != NULL){
            prev = slowPtr;
            slowPtr = slowPtr->next;
            fastPtr  = fastPtr->next->next;
      }
      prev->next = NULL;
      Node* l1 = mergeSort(head);
      Node* l2 = mergeSort(slowPtr);
      return merge(l1, l2);
}
Node* merge(Node* l1, Node* l2){
      Node dummy(1);
      Node* p = &dummy;
      while (l1 && l2){
            if (l1->data < l2->data){
                  p->next = l1;
                  l1 = l1->next;
            }
            else {
                  p->next = l2;
                  l2 = l2->next;
            }
            p = p->next;
      }
      if (l1)
            p->next = l1;
      if (l2)
            p->next = l2;
      return dummy.next;
```

```
}
```

**Complexity:**

- Time: O(N log N)
- Space: O(log N)