EASY :

1. Palindrome linked list

```
class Solution {
public:
    bool isPalindrome(ListNode* head)
    {
       ListNode *l = head;
        ListNode *right = head;
        ListNode *prev = NULL;
        ListNode *tmp;
        while(right !=NULL && right->next!=NULL)
        {
            right = right->next->next;
            tmp = l->next;
            l->next = prev;
            prev = l;
            l = tmp;
        }
        if(right!=NULL)
        l = l->next;
        while(prev && l)
        {
            if(prev->val==l->val)
            {
                prev=prev->next;
                l=l->next;
            }
            else
            return false;
        }
        return true;
    }
};
```

2. Check if circular linked list

```
bool isCircular(struct Node *head){
        struct Node *slow=head;
        struct Node *fast=head;

        while(fast&&fast->next)
        {
         slow=slow->next;
          fast=fast->next->next;
          if(fast==slow)
          return true;
      }
      return false;
}
```

MEDIUM :

1. Reorder linked list

```
class Solution {
public:
     ListNode* pre;
     void reorder(ListNode* node)
     {
        if(node==0)
        return;

        reorder(node->next);

        if(!pre)
        return;

         else if(pre==node || pre->next==node)
        {
            node->next=0;
            pre=0;
            return;
        }

        node->next=pre->next;
        pre->next=node;
        pre=node->next;
    }

    void reorderList(ListNode* head)
    {
        pre=head;
        reorder(head);
    }
};
```

2. Remove nth node from the end

```
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n)
    {
        ListNode *res=head;
        ListNode *ret=head;
        int c=0;
        if(c==n)
        return ret->next;

        while(head)
        {
            c++;
            head=head->next;
        }
        c=c-n-1;
        c--;
        while(c>=0)
        {
            c--;
            res=res->next;
        }
        res->next=res->next->next;
        return ret;
     }
};
```

3. Even odd linked list

```
class Solution {
public:
    vector<int>ans;
    vector<int>ans1;
    int k=0;
    void solve(ListNode *head)
    {
        while(head)
        {
            if(k%2==0)
            ans1.push_back(head->val);
            else
            ans.push_back(head->val);
            head=head->next;
            k++;
        }
    }
    ListNode* oddEvenList(ListNode* head)
    {
        if(head==0 || head->next==0)
        return head;
        solve(head);

        ListNode *start=new ListNode(ans1[0]);
        ListNode *st=start;
        for(int i=1;i<ans1.size();i++)
        {
            ListNode *temp=new ListNode(ans1[i]);
            start->next=temp;
            start=start->next;
        }
        for(int i=0;i<ans.size();i++)
        {
            ListNode *temp=new ListNode(ans[i]);
            start->next=temp;
            start=start->next;
        }
        return st;
        }
};
```

4. Sort linked list using merge sort

```
class Solution {
public:
    void sortList(ListNode** head )
    {
        ListNode* first;
        ListNode* second;
        ListNode* cur=*head;
        if(cur==0 ||cur->next==0)
        return ;
        break_in_half(cur,&first,&second);
        sortList(&first);
        sortList(&second);
        *head=merge_sort(first,second);
    }
    void break_in_half(ListNode* cur,ListNode **first,ListNode **second)
    {
        ListNode* slow=cur;
        ListNode* fast=cur->next;
        while(fast!=0)
        {
            fast=fast->next;
            if(fast!=0)
            {
                fast=fast->next;
                slow=slow->next;
            }
        }
        *first=cur;
        *second=slow->next;
        slow->next=0;
    }
    ListNode* merge_sort(ListNode* l1,ListNode* l2)
    {
        if(l2==0)
        return l1;
        if(l1==0)
        return l2;
        ListNode* head;
        if(l1->val<l2->val)
        {
            head=l1;
            head->next=merge_sort(l1->next,l2);
        }
        else
        {
            head=l2;
            head->next=merge_sort(l1,l2->next);
        }
        return head;
    }
    ListNode* sortList(ListNode* head)
    {
        sortList(&head);
        return head;
    }
};
```

5. Strating point of loop in linked list

```
class Solution {
public:
    ListNode *detectCycle(ListNode *head)
    {
       if(!head)
        return 0;
        ListNode* fast = head;
        ListNode *slow = head;

        while(fast->next and fast->next->next)
        {
            fast = fast->next->next;
            slow = slow->next;

            if(fast == slow)
            {
                slow = head;
                while(fast != slow)
                {
                    fast = fast->next;
                    slow = slow->next;
                }
                 return fast;
               }
           }
        return 0;
    }
};
```

6. Copy list with random pointer

```
class Solution {
public:
    Node* copyRandomList(Node* head)
    {
        unordered_map<Node*,Node*>mp1;
        Node *start = head;
        Node *new_node = new Node(0);
        Node *rtn=new_node;
        mp1[0] = 0;
        while(start)
        {
            Node *temp = new Node(start->val);
            new_node->next = temp;
            new_node = new_node->next;
            mp1[start] = new_node;
            start = start->next;
        }
        start = head;
        new_node = head;
        while(start)
        {
            new_node->random = mp1[start->random];
            new_node = new_node->next;
            start = start->next;
        }


        return rtn->next;
    }
};
```

7. Partition list

```
class Solution {
public:
    Node* copyRandomList(Node* head)
    {
        unordered_map<Node*,Node*>mp1;
        Node *start = head;
        Node *new_node = new Node(0);
        Node *rtn=new_node;
        mp1[0] = 0;
        while(start)
        {
            Node *temp = new Node(start->val);
            new_node->next = temp;
            new_node = new_node->next;
            mp1[start] = new_node;
            start = start->next;
        }
        start = head;
        new_node = head;
        while(start)
        {
            new_node->random = mp1[start->random];
            new_node = new_node->next;
            start = start->next;
        }


        return rtn->next;
    }
};
```

HARD :

1. Reverse linked list in k groups

```
class Solution {
public:
    ListNode *reverse(ListNode *head, int k)
    {
        ListNode *curr = head;
        ListNode *prev = 0;
        ListNode *next;
        while(curr && k)
        {
            next = curr -> next;
            curr -> next = prev;
            prev = curr;
            curr = next;
            k--;
        }
        return prev;
    }
    ListNode* reverseKGroup(ListNode* head, int k)
    {
        if(k == 1)
        return head;

        int c=0;
         ListNode *curr = head;
        ListNode *next = 0;

        while(curr && c<k)
        {
            curr = curr -> next;
            c++;
        }
        if(c<k)
        return head;
        next = curr;

        ListNode *n_head = reverse(head,k);
        head -> next = reverseKGroup(next,k);
        return n_head;
    }
};
```