1.Reverse a linked list:

Method 1:Iterative…

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

Method 2:Recursive

```
class Solution
{
   public:
   Node* head=NULL;
   struct Node* reverseList(struct Node *node){
      Node* t=reverse(node);
      return head;
   }
   //Function to reverse a linked list.
   struct Node* reverse(struct Node *node)
   {
```

```
        if (node == NULL)
            return NULL;
        if (node->next == NULL) {
            head = node;
            return node;
        }
        Node* node1 = reverse(node->next);
        node1->next = node;
        node->next = NULL;
        return node;
    }

};
```

2.Reverse a Linked List in groups of given size.

```
class Solution
{
    public:
    struct node *reverse (struct node *head, int k)
    {
        int count=0;
        node* prev=NULL;
        node* next=NULL;
        node* curr=head;
```

```
        while(curr && count<k){
            next=curr->next;
            curr->next=prev;
            prev=curr;
            curr=next;
            count++;
        }
        if(next!=NULL)head->next=reverse(next,k);

        return prev;
    }
};
```

4.Detect Loop in linked list :

Method 1:unordered_set

Method 2:flag in the struct body.

url :geeksforgeeks.org/detect-loop-in-a-linked-list/

Method 3:Sptr and fptr

```
class Solution
{
    public:
    bool detectLoop(Node* head)
    {
        Node* sptr=head;
        Node* fptr=head;
```

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

4.Remove loop in LL:

Method 1:Flag

```
#include<bits/stdc++.h>
using namespace std;
struct Node
{
    int val;
    Node* next;
    bool flag;
    Node(int data)
    {
        val=data;
        next=NULL;
        flag=0;
    }
};
void detect_remove_loop_ll(Node* head)
{
```

```
    Node* prev=NULL;
    while(head){
        if(head->flag==1){
            prev->next=NULL;return;
        }
        head->flag=1;
        prev=head;
        head=head->next;
    }
return;
}
int main(){
    Node* node=new Node(4);
    node->next=new Node(3);
    node->next->next=new Node(2);
    node->next->next->next=new Node(1);
    node->next->next->next->next=node->next;

    detect_remove_loop_ll(node);
    cout<<node->next->next->next->next->val;}
```

Method 2:Fptr and Sptr

Method 2 (Better Solution)

1.This method is also dependent on Floyd's Cycle detection algorithm.
2.Detect Loop using Floyd's Cycle detection algorithm and get the pointer to a loop node.
3.Count the number of nodes in loop. Let the count be k.

4.Fix one pointer to the head and another to a kth node from the head.
5.Move both pointers at the same pace, they will meet at loop starting node.
6.Get a pointer to the last node of the loop and make next of it as NULL.

Method 3:

We do not need to count number of nodes in Loop. After detecting the loop, if we start slow pointer from head and move both slow and fast pointers at same speed until fast don't meet, they would meet at the beginning of the loop.

5.Remove duplicate element from sorted Linked List:

```
void removeDuplicates(struct Node* head)
{
   if(head==NULL)return;
   while(head->next){
      if(head->data==head->next->data)
      {
         struct Node* temp=head->next;
         head->next=head->next->next;
         free(temp);
      }
      else
      head=head->next;
```

```
    }
    return;
}
```

6.Remove duplicates from an unsorted linked list :

```
class Solution
{
    public:
    //Function to remove duplicates from unsorted linked list.
    Node* removeDuplicates( Node *head)
    {
     unordered_set<int> m;
     Node* curr=head;
     Node* prev=NULL;
     while(curr){
         if(m.find(curr->data)!=m.end()){
             prev->next=curr->next;
             delete(curr);
         }
         else{
             m.insert(curr->data);
             prev=curr;
         }
         curr=prev->next;
     }
    }
```

```
};
```

7.Add 1 to a number represented as linked list:

```
class Solution
{
    public:
    Node* reverse(Node* head){
        Node* prev=NULL;
        Node* curr=head;
        Node* t=NULL;
        while(curr){
            t=curr->next;
            curr->next=prev;
            prev=curr;
            curr=t;
        }
        return prev;
    }
    Node* addOne(Node *head)
    {
        bool flag=true;
        head=reverse(head);
        Node* curr=head;
        while(curr && flag){
            if(curr->data==9 && curr->next==NULL){
                curr->data=1;
                Node* temp=new Node(0);
                temp->next=head;
                head=temp;
```

```
            break;
        }
        else if(curr->data==9 ){
            curr->data=0;
            curr=curr->next;
        }

        else{
            curr->data=curr->data+1;
            break;
        }
    }
    head=reverse(head);
  }
};
```

8.Add two numbers represented by linked lists

```
class Solution
{
   public:
   Node* reverse(Node* head){
      Node* prev=NULL;
      Node* curr=head;
      Node* t=NULL;
      while(curr){
         t=curr->next;
         curr->next=prev;
         prev=curr;
         curr=t;
```

```
        }
        return prev;
    }
    //Function to add two numbers represented by linked list.
    struct Node* addTwoLists(struct Node* first, struct Node*
second)
    {
        first=reverse(first);
        second=reverse(second);
        Node* curr=NULL;
        Node* res=NULL;
        int c=0;
        while(first || second){
            int
sum=c+(first?first->data:0)+(second?second->data:0);
            c=(sum>=10?1:0);
            int val=sum%10;
            Node* temp=new Node(val);
            if(res==NULL)res=temp;
            else curr->next=temp;
            curr=temp;
            if(first)first=first->next;
            if(second)second=second->next;
        }
        if(c>0){
            Node* temp=new Node(c);
            curr->next=temp;
            temp=curr;
        }
        res=reverse(res);
```

```
        return res;
    }
};
```

9.Intersection of two sorted Linked lists :

```
Node* findIntersection(Node* head1, Node* head2)
{
    Node* head=NULL;
    Node* curr=NULL;
    while(head1 && head2){
        if(head1->data==head2->data){
            Node* temp=new Node(head1->data);
            if(head==NULL){
                head=temp;
                curr=temp;
            }
            else{
                curr->next=temp;
                curr=temp;
            }
            head1=head1->next;
            head2=head2->next;
        }
        else{
            if(head1->data>head2->data){
                head2=head2->next;
            }
            else{
                head1=head1->next;
```

```
            }
        }
        //return head;
    }
    return head;
}
```

10.Merge sort:

a.Using array:

```
void merge(int arr[], const int l,const  int m,const int r)
    {
        int count1=m-l+1;
        int count2=r-m;
        int *arr1=new int[count1];
        int *arr2=new int[count2];

        for(int i=0;i<count1;i++)arr1[i]=arr[l+i];
        for(int i=0;i<count2;i++)arr2[i]=arr[m+1+i];
         int i=0,j=0,k=l;
         while(i<count1 && j<count2){
            if(arr1[i]<=arr2[j]){
                arr[k]=arr1[i];i++;
            }
            else{
                 arr[k]=arr2[j];j++;
            }
```

```
            k++;
        }
        while(i<count1){
            arr[k]=arr1[i];i++;k++;
        }
        while(j<count2){
            arr[k]=arr2[j];j++;k++;
        }
       return;
        // Your code here
    }
    public:
    void mergeSort(int arr[], int l, int r)
    {
        if (l>=r)return ;
        int mid=l+(r-l)/2;
        mergeSort(arr,l,mid);
        mergeSort(arr,mid+1,r);
        merge(arr,l,mid,r);
    }
```

b.Using Lists:

```
class Solution{
  public:
  void middle(Node* curr,Node** first,Node** second){
      Node* sptr=curr;
      Node* fptr=curr->next;
      while(fptr){
          fptr=fptr->next;
```

```
        if(fptr){
            sptr=sptr->next;
            fptr=fptr->next;
        }
    }
    *first=curr;
    *second=sptr->next;
    sptr->next=NULL;
}
Node* mergeboth(Node* first,Node* second){
    Node* result=NULL;
    if(!first)return second;
    if(!second)return first;

    if(first->data<=second->data){
        result=first;
        result->next=mergeboth(first->next,second);
    }
    else{
         result=second;
        result->next=mergeboth(first,second->next);
    }
    return result;

}
void mergesort(Node** head){
    Node* first=NULL;
    Node* second=NULL;
    Node* curr=*head;
    if(!curr or !curr->next)return ;
```

```
      middle(curr,&first,&second);

      mergesort(&first);
      mergesort(&second);
     *head= mergeboth(first,second);
      //return *head;
  }
    //Function to sort the given linked list using Merge Sort.
    Node* mergeSort(Node* head) {
      mergesort(&head);
      return head;
    }
};
```

11.Quick sort:

a.Using arrays:

```
void quickSort(int arr[], int low, int high)
{
    if(low<high){
        int p=partition(arr,low,high);
        quickSort(arr,low,p-1);
        quickSort(arr,p+1,high);
    }
}

int partition (int arr[], int low, int high)
{
```

```
  int pivot=arr[high];
  int i=low-1;
  for(int j=low;j<=high-1;j++){
     if(arr[j]<pivot){
        i++;
        int temp=arr[i];
        arr[i]=arr[j];
        arr[j]=temp;
     }
  }
  i++;
  int temp=arr[i];
  arr[i]=pivot;
  arr[high]=temp;
  return i;
}
```

b.Using Linked List:

13.Find the middle Element of a linked list.

Given the head of a singly linked list, return the middle node of the linked list.
If there are two middle nodes, return the second middle node.

```
class Solution {
public:
   ListNode* middleNode(ListNode* head) {
```

```
        ListNode* s=head;
        ListNode* f=head;
        while(f){
            f=f->next;
            if(f){
                s=s->next;
                f=f->next;
            }
        }
        return s;
    }
};
```

14.Check if a linked list is a circular linked list.

```
bool isCircular(struct Node *head){
    if(!head)return true;
    struct Node* curr=head;
    while(curr){
        curr=curr->next;
        if(curr==head)return true;
    }
    return false;
}
```

15.Split a Circular linked list into two halves.

```
void splitList(Node *head, Node **head1_ref, Node
**head2_ref)
{
    Node* s=head;
    Node* f=head->next;
    Node* curr=head;
    Node* prev=NULL;
    while(curr){
        prev=curr;
        curr=curr->next;
        if(curr==head){
            prev->next=NULL;
        }
    }
    curr=head;
    while(f){

        f=f->next;
        if(f){
            f=f->next;
            s=s->next;
        }
        //if(f)prev=f;


    }
     Node* temp=s->next;
        s->next=head;
```

```
        prev->next=temp;
        *head1_ref=head;
        *head2_ref=temp;
}
```

16.Write a Program to check whether the Singly Linked list is a palindrome or not.

Method 1:

```
class Solution{
  public:
    //Function to check whether the list is palindrome.
    bool isPalindrome(Node *head)
    {
        stack<int> s;
        Node* curr=head;
        while(curr){
            s.push(curr->data);
            curr=curr->next;
        }
        while(head){
            int i=s.top();
            s.pop();
            if(head->data!=i)return 0;
            head=head->next;
        }
        return 1;
    }
```

};
Complexity:time=O(N) Space=O(N)
METHOD 2 (By reversing the list)
This method takes O(n) time and O(1) extra space.
1) Get the middle of the linked list.
2) Reverse the second half of the linked list.
3) Check if the first half and second half are identical.
4) Construct the original linked list by reversing the second half again and attaching it back to the first half

To divide the list into two halves, method 2 of this post is used.

When a number of nodes are even, the first and second half contain exactly half nodes. The challenging thing in this method is to handle the case when the number of nodes is odd. We don't want the middle node as part of the lists as we are going to compare them for equality. For odd cases, we use a separate variable 'midnode'.

Complexity:Time=O(N) space=O(1)

16.Deletion and Reverse in circular Linked List: First delete and then reverse

```
void deleteNode(struct Node **head, int key)
{
   Node* curr=*head;
```

```
if(!(curr) || !(curr->next))
{
   *head=NULL;return;
}
while(curr->next->data!=key){
   curr=curr->next;
}
Node* t=curr->next;
curr->next=t->next;
free(t);
}

/* Function to reverse the linked list */
 void reverse(struct Node** head)
{

   Node* curr=*head;
   Node* prev=NULL;
   Node* t=NULL;

      t=curr->next;
      curr->next=prev;
      prev=curr;
      curr=t;

   while(curr!=*head){
      t=curr->next;
      curr->next=prev;
      prev=curr;
      curr=t;
```

```
    }
    curr->next=prev;
    *head=prev;

}
```

17.Reverse a Doubly Linked List

```
struct Node* reverseDLL(struct Node * head)
{
   struct Node* curr=head;
    struct Node* t=NULL;
    struct Node* prev=NULL;
    while(curr){
        t=curr->next;
        curr->next=prev;
        prev=curr;
        curr->prev=t;
        curr=t;
    }
    return prev;
}
```

18.Find pairs with a given sum in a DLL.

```
#include<bits/stdc++.h>
using namespace std;
class Node{
    public:
```

```
    int data;
    Node* next;
    Node* prev;
    Node(int val){
        data=val;next=prev=NULL;
    }
    void pairsDll(Node* head,int x){
       Node* last=head;
        Node* first=head;
        while(last->next)last=last->next;
        while(first!=last && last->next!=first)
        {
            if(first->data + last->data ==x){

cout<<"("<<first->data<<","<<last->data<<")"<<endl;
                first=first->next;
                last=last->prev;
            }
            else if(first->data + last->data >x)last=last->prev;
            else first=first->next;
        }
 }
 int sum(int d){return d+d;}
};

int main(){
  Node* head=new Node(1);
  head->prev=NULL;
   head->next=new Node(2);
   head->next->prev=head;
```

```
  head->next->next=new Node(3);
  head->next->next->prev=head->next;
  head->next->next->next=new Node(4);
  head->next->next->next->prev=head->next->next;

  head->next->next->next->next=new Node(5);

head->next->next->next->next->prev=head->next->next->next
;
  int x=6;
  Node* ob;
  ob->pairsDll(head,x);

   return 0;
}
```

19.Count triplets in a sorted DLL whose sum is equal to given value “X”.

```
#include<bits/stdc++.h>
using namespace std;
class Node{
   public:
   int data;
   Node* next;
   Node* prev;
   Node(int val){
      data=val;next=prev=NULL;
   }
   void pairsDll(Node* head,int x){
```

```
      Node* last=head;
       Node* first=head;
       while(last->next)last=last->next;
       while(first!=last && last->next!=first && first->next!=last)
       {
          if(first->data+first->next->data + last->data ==x){

cout<<"("<<first->data<<","<<first->next->data<<","<<last->d
ata<<")"<<endl;
              first=first->next;
              last=last->prev;
          }
          else if(first->data + first->next->data+last->data
>x)last=last->prev;
          else first=first->next;
       }
 }};

int main(){
  Node* head=new Node(1);
  head->prev=NULL;
   head->next=new Node(2);
   head->next->prev=head;
   head->next->next=new Node(3);
   head->next->next->prev=head->next;
   head->next->next->next=new Node(4);
   head->next->next->next->prev=head->next->next;

   head->next->next->next->next=new Node(5);
```

```
head->next->next->next->next->prev=head->next->next->next
;
   int x=8;
   Node* ob;
   ob->pairsDll(head,x);

    return 0;
}
```

20.Sort a “k”sorted Doubly Linked list.[Very IMP]

```
struct compare {
    bool operator()(struct Node* p1, struct Node* p2)
    {
        return p1->data > p2->data;
    }
};

void sort_k_Dll(Node** head,int k){
    priority_queue<Node* ,vector<Node*>,compare> q;
    Node* curr=*head;
    Node* prev1=NULL;
    int c=0;
    while(curr){
        if(q.size()<=k){
            q.push(curr);
            curr=curr->next;
    else{
```

```
    Node* t=q.top();q.pop();
    if(c==0){*head=t;c=1;}
    t->prev=prev1;
    t->next=q.top();
    prev=t;
    q.push(curr);
    curr=curr->next;
    }}
    while(!q.empty()){
        Node* t=q.top();q.pop();
        if(c==0){*head=t;c=1;}
    t->prev=prev1;
    if(q.top())t->next=q.top();
    else t->next=NULL;
    prev=t;
    }
 }
};
```

21.Rotate DoublyLinked list by N nodes.

```
class Solution {
public:
    Node *rotateDLL(Node *start,int p)
    {
        Node* curr=start;
        while(curr->next)curr=curr->next;
        int count=1;
        Node* mid=start;
```

```
        while(count<p){
            mid=mid->next;
            count++;
        }

        Node* t=mid->next;
        mid->next=NULL;
        curr->next=start;
        t->prev=NULL;
        start=t;


    return start;}
};
```

22.Can we reverse a linked list in less than2 O(n)?

It is not possible to reverse a simple singly linked list in less than O(n). A simple singly linked list can only be reversed in O(n) time using recursive and iterative methods.

A doubly linked list with head and tail pointers while only requiring swapping the head and tail pointers which require lesser operations than a singly linked list can also not be done in less than O(n) since we need to traverse till the end of the list anyway to find the tail node.

23.Flatten a Linked List

```
Node* merge(Node* first,Node* second){
      Node* result=NULL;
      if(!first)return second;
      if(!second)return first;

      if(first->data<=second->data){
         result=first;
         result->bottom=merge(first->bottom,second);
      }
      else{
          result=second;
         result->bottom=merge(first,second->bottom);
      }
      result->next=NULL;
      return result;

  }

Node *flatten(Node *root)
{
   if(!root || !root->next)return root;
   return merge(root,flatten(root->next));

}
```

Time Complexity: O(N*N*M) – where N is the no of nodes in main linked list (reachable using right pointer) and M is the no of node in a single sub linked list (reachable using down pointer).
Explanation: As we are merging 2 lists at a time,

After adding first 2 lists, time taken will be O(M+M) = O(2M).
Then we will merge another list to above merged list -> time = O(2M + M) = O(3M).
Then we will merge another list -> time = O(3M + M).
We will keep merging lists to previously merged lists until all lists are merged.
Total time taken will be O(2M + 3M + 4M + …. N*M) = (2 + 3 + 4 + … + N)*M
Using arithmetic sum formula: time = O((N*N + N – 2)*M/2)
Above expression is roughly equal to O(N*N*M) for large value of N

Space Complexity: O(N*M) – because of the recursion. The recursive functions will use recursive stack of size equivalent to total number of elements in the lists, which is N*M.

Method 2:Priority queues

```
struct mycomp {
    bool operator()(Node* a, Node* b)
    {
```

```
        return a->data > b->data;
    }
};


void flatten(Node* root)
{
    priority_queue<Node*, vector<Node*>, mycomp> p;
  //pushing main link nodes into priority_queue.
    while (root!=NULL) {
        p.push(root);
        root = root->next;
    }

    while (!p.empty()) {
      //extracting min
        auto k = p.top();
        p.pop();
      //printing  least element
        cout << k->data << " ";
        if (k->bottom)
            p.push(k->bottom);
    }}
```

Time Complexity: O(N*M*log(N)) – where N is the no of nodes in main linked list (reachable using next pointer) and M is the no of node in a single sub linked list (reachable using bottom pointer).

Space Complexity: O(N)-where N is the no of nodes in main linked list (reachable using next pointer).

24.Sort a LL of 0's, 1's and 2's

```
class Solution
{
    public:
    //Function to sort a linked list of 0s, 1s and 2s.
    Node* segregate(Node *head) {
        Node* curr=head;int a=0,b=0,c=0;
       while(curr){
           if(curr->data==0)a++;
           else if(curr->data==1)b++;
           else c++;

           curr=curr->next;
       }
       curr=head;
       while(curr){
           if(a){curr->data=0;a--;}
           else if(b){curr->data=1;b--;}
           else {curr->data=2;c--;}
           curr=curr->next;
    }
    return head;
    }
};
```

25.Clone a linked list with next and random pointer:

```
class Solution
{
   public:
   Node *copyList(Node *head)
   {
      Node* curr=head;
      while(curr)
      {
         Node* t=curr->next;
         curr->next=new Node(curr->data);
         curr->next->next=t;
         curr=t;
      }
      curr=head;
      while(curr){
         curr->next->arb=curr->arb?curr->arb->next:curr->arb;
         curr=curr->next->next;
      }
      Node* original=head;
      Node* copy=head->next;
      Node* t=copy;
      while(original && copy){
         original->next=original->next->next;

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

```
        }
        return t;
    }

};
```

26.Merge K sorted Linked list

```
class Solution{
  public:
    //Function to merge K sorted linked list.
    Node* merge(Node* first,Node* second){
        if(!first)return second;
        if(!second)return first;
        Node* res=NULL;

        if(first->data<=second->data){
            res=first;
            res->next=merge(first->next,second);
        }
        else {
            res=second;
            res->next=merge(first,second->next);
        }
        return res;
    }
    Node * mergeKLists(Node *arr[], int K)
    {
          int i=1;
          while(i<K)
```

```
        {
          arr[0]= merge(arr[0],arr[i]);
          i++;
        }
        return arr[0];
    }
};
```

27.Multiply 2 no. represented by LL:

```
long long  multiplyTwoLists (Node* l1, Node* l2)
{
 int N=1000000007;
 long int mult1=0,mult2=0;
  while(l1){
      mult1=(mult1*10)%N+l1->data;
      l1=l1->next;
  }
  while(l2){
      mult2=(mult2*10)%N+l2->data;
      l2=l2->next;
  }
  return (mult1*mult2)%N;
}
```

28.Delete nodes which have a greater value on right side

Note:Haven’t accepted all cases

```
class Solution
{
   public:
   Node *compute(Node *head)
   {
     Node* curr=head;
     Node* t=NULL;
     Node* prev=NULL;
     while(curr->next){
        if(curr->data<curr->next->data){
           if(curr==head){
              t=curr;
              curr=curr->next;
              head=curr;
              t->next=NULL;
              free(t);
              prev=curr;
           }
           else {
              t=curr;
              prev->next=curr->next;
              curr->next=NULL;
              free(t);
              curr=prev;
           }
        }
        else{
        prev=curr;
```

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

};
```

29.Segregate even and odd nodes in a Linked List:

```
class Solution{
public:
    Node* divide(int N, Node *head){
        Node* even=NULL;
        Node* odd=NULL;
        Node* e=NULL;
        Node* o=NULL;
        //Node* curr=head;
        while(head){
            if(head->data%2==0){
                if(even == NULL){even=head;e=head;}
                else{
                    e->next=head;e=e->next;
                }

            }
```

```
            else{
                if(odd == NULL){odd=head;o=head;}
                else{
                    o->next=head;o=o->next;
                }

            }
            head=head->next;
        }
        if(e)e->next=odd;
        if(o)o->next=NULL;
        if(even)return even;
        return odd;

    }
};
```

30.Program for n’th node from the end of a Linked List:

```
int getNthFromLast(Node *head, int n)
{
     int c1=0;
    struct Node* curr=head;
    while(curr){
        c1++;
        curr=curr->next;
    }
    if(c1<n)return -1;
```

```
    curr=head;
    c1=c1-n;
    while(c1--){
        curr=curr->next;
    }
    return curr->data;
}
```