LINKED LIST

A general approachg to pass all test cases in linked list is
1. If head == null || head->next == null return
2. Check the written code if it works for botjh odd length and even length LLs

Always create a dummy node if required because in many problems we use prev and next pointers and using a dummy node prevents writing hard edge cases coz if we dont use dummy node we would moslty set prev to null and this increases chances of nullptr exception as well.

Linked List

```
void fold(Node *head){
    if(head == NULL || head->next == NULL)
        return;

    //Find the middle of the linked list
    Node *slow = head;
    Node *fast = head;
    while(fast->next && fast->next->next){
        slow = slow->next;
        fast = fast->next->next;
    }

    //Reverse second half AND BREAK THE LINKED LIST INTO TWO SEPARATE PARTS
    Node *rhead = reverse(slow->next);
    slow->next = nullptr;

    // Fold the linked list
    Node *node1 = head;
    Node *node2 = rhead;
    Node *n1Nxt , *n2Nxt = node2->next;
    while(node2 != nullptr){
        n1Nxt = node1->next;
        n2Nxt = node2->next;

        node1->next = node2;
        node2->next = n1Nxt;

        node1 = n1Nxt;
        node2 = n2Nxt;
    }
}
```

```
Input
    1->2->3->4->5->6->7->null
Output
    1->7->2->6->3->5->4->null
```

Fold of a linked list

```
void unFold(Node *head){
    if(head == NULL || head->next == NULL)
        return;

    Node *tail = head;
    Node *nHead = head->next;

    Node *node1 = head;
    Node *node2 = nHead;
    while(node2){
        Node *n1Nxt = node2->next;
        Node *n2Nxt = node2->next != nullptr ? node2->next->next : node2->next; //Edge

        node1->next = n1Nxt;
        node2->next = n2Nxt;

        node1 = n1Nxt;
        node2 = n2Nxt;

        if(node1) tail = node1;
    }
    tail->next = reverse(nHead);
}
```

unfold a LL

```
Node* MergeLL(Node *head1, Node *head2){  //T=O(n1+n2) S=O(1)
    if(head1 == NULL)
        return head2;
    if(head2 == NULL)
        return head1;

    Node *c1 = head1;
    Node *c2 = head2;
    Node *dummy = new Node(-1);
    Node *prev = dummy;

    while(c1 && c2){
        if(c1->data  < c2->data){
            prev->next = c1;
            c1 = c1->next;
        }
        else{
            prev->next = c2;
            c2 = c2->next;
        }
        prev = prev->next;
    }
    prev->next = c1 != nullptr ? c1 : c2;
    return dummy->next;
}
```

Merge 2 sorted LL

```
Node* MergeLL(Node *head1, Node *head2){  //T=O(n1+n2) S=O(1)
    if(head1 == NULL)
        return head2;
    if(head2 == NULL)
        return head1;

    Node *c1 = head1;
    Node *c2 = head2;
    Node *dummy = new Node(-1);
    Node *prev = dummy;

    while(c1 && c2){
        if(c1->data  < c2->data){
            prev->next = c1;
            c1 = c1->next;
        }
        else{
            prev->next = c2;
            c2 = c2->next;
        }
        prev = prev->next;
    }
    prev->next = c1 != nullptr ? c1 : c2;
    return dummy->next;
}

Node* mergeKSLL(vector<Node*> lists, int si, int ei){
    if(si > ei)
        return nullptr;
    if(si == ei)
        return lists[si];

    int mid = (si + ei) / 2;
    Node *l1 = mergeKSLL(lists, si, mid);
    Node *l2 = mergeKSLL(lists, mid+1, ei);
    return MergeLL(l1, l2);
}

Node* mergeKSLL(vector<Node*> lists){
    if(lists.size() == 0)
        return nullptr;
    return mergeKSLL(lists, 0, lists.size() - 1);
}
```

Merge K Sorted Lists using Merge Sort algo

```
Node* mergeKSLL(vector<Node*> lists){
    if(lists.size() == 0)
        return nullptr;

    priority_queue< Node*, vector<Node*> , myComparator > pq;
    //Push head of each Linked List into PQ (if not null)
    for(Node *n : lists){
        if(n){
            pq.push(n);
        }
    }
    //Sort
    Node *dummy = new Node(-1);
    Node *prev = dummy;
    while(!pq.empty()){
        Node *node = pq.top();
        pq.pop();

        prev->next = node;
        prev = prev->next;

        if(node->next) pq.push(node->next);
    }
    prev->next = nullptr;
    return dummy->next;
}
```

Merge K Sorted Lists using Priority Queue

```cpp
Node* MergeLL(Node *head1, Node *head2){  //T=O(n1+n2) S=O(1)
    if(head1 == NULL)
        return head2;
    if(head2 == NULL)
        return head1;

    Node *c1 = head1;
    Node *c2 = head2;
    Node *dummy = new Node(-1);
    Node *prev = dummy;

    while(c1 && c2){
        if(c1->data  < c2->data){
            prev->next = c1;
            c1 = c1->next;
        }
        else{
            prev->next = c2;
            c2 = c2->next;
        }
        prev = prev->next;
    }
    prev->next = c1 != nullptr ? c1 : c2;
    return dummy->next;
}

Node *middle(Node *head){
    if (head == nullptr || head->next == nullptr)
        return head ;

    Node *slow = head ;
    Node *fast = head ;
    while(fast->next != nullptr && fast->next->next != nullptr){
        slow = slow->next ;
        fast = fast->next->next ;
    }
    return slow;
}

Node* MergeSort(Node *head){
    if(head == nullptr)
        return nullptr;
    if(head->next == nullptr)
        return head;

    //Dividing linked list into two parts
    Node *mid = middle(head);
    Node *nHead = mid->next;
    mid->next = nullptr;

    //Merge left sorted and right sorted LL
    Node *left = MergeSort(head);
    Node *right = MergeSort(nHead);
    return MergeLL(left, right);
}
```

Merge Sort of LL

```cpp
static Node *th = NULL; //temporary head
static Node *tt = NULL; //temporary tail
void addFirst(Node *node){
    if(th == nullptr){
        th = node;
        tt = node;
    }
    else{
        node->next = th;
        th = node;
    }
}

Node* reverse(Node *head){
    Node *curr = head;
    while(curr){
        Node *next = curr->next;
        curr->next = nullptr;
        addFirst(curr);
        curr = next;
    }
    return th;
}
```

Reverse using Add First

```cpp
Node *reverse(Node *head){
    Node *curr = head, *prev = nullptr, *next;
    while(curr){
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
    return prev;
}
```

Normal Reversal

Reverse using Add Firsrt can be helpful in problems which requre a reference to both head and tail fo reversed list.
Eg Reverse in K groups

```
1. Traverse till we are one node before REV RANGE
       link  curr
         |    |
         v    v
    8 -> 8 -> 14 -> 1-> 10 -> 12 -> null

2. Reverse nodes in range
       link   tt        th    curr
         |    |         |      |
         v    v         v      v
    8 -> 8 -> 10 -> 1-> 14 -> 12 -> null

3. link->next = th
   tt->next = curr
```

```cpp
Node *reverseInRange(Node *head, int si, int ei){
    if(head == nullptr || head->next == nullptr)
        return head;

    Node *curr = head;
    Node *link = nullptr;
    int revRange = ei-si+1;
    //Iterate till link points to si ie node before reverse range
    while(--si > 0){
        link = curr;
        curr = curr->next;
    }
    Node *prev = nullptr;
    //Reverse nodes in given range
    while(curr && --revRange >= 0){
        Node *next = curr->next;
        curr->next = nullptr;
        addFirst(curr);
        curr = next;
    }
    if(link){
        link->next = th;
    }
    tt->next = curr;
    return link != nullptr ? head : th;
}
```

Reverse in range

```cpp
static Node *th = NULL; //temporaty head
static Node *tt = NULL; //temporaty tail
void addFirst(Node *node){
    if(th == nullptr){
        th = node;
        tt = node;
    }
    else{
        node->next = th;
        th = node;
    }
}

int length(Node *head){
    int count = 0;
    Node *temp = head;
    while(temp){
        temp = temp->next;
        count++;
    }
    return count;
}

Node* reverseKnodes(Node *head, int K){
    if(K==0 || K==1)
        return head;

    int len = length(head);
    Node *curr = head;
    Node *oh = nullptr, *ot = nullptr; //Original head and Original tail
    while(len >= K){
        int tempk = K;
        //Reverse K nodes
        while(tempk-- > 0){
            Node *next = curr->next;
            curr->next = nullptr;
            addFirst(curr);
            curr = next;
        }
        //Linking groups
        if(oh == nullptr){
            oh = th;
            ot = tt;
        }
        else{
            ot->next = th;
            ot = tt;
        }
        th = nullptr;
        tt = nullptr;
        len -= K;
    }
    ot->next = curr;
    return oh;
}
```

Reverse in k group

```cpp
Node* CloneLinkedList(Node *head){
    Node *node = head;
    Node *clone = NULL;
    //Step 1 -> Introducing new nodes of clone linked list btw original linked list
    while(node != NULL){
        clone = new Node(node->data);
        clone->next = node->next;
        clone->random = NULL;

        node->next = clone;
        node = node->next->next;
    }
    //Step 2 -> Updating random pointers of cloned linked list
    node = head;
    while(node != NULL){
        if(node->random != NULL)
            node->next->random = node->random->next;
        node = node->next->next;
    }
    //Step 3 -> Separating original linked list and cloned linked list
    Node *temp;
    node = head;
    clone = node->next;
    Node *cloneNode = clone;
    while(node != NULL){
        temp = cloneNode->next;
        if(temp != NULL)
            cloneNode->next = temp->next;
        else
            cloneNode->next = NULL;
        node->next = temp;
        node = node->next;
        cloneNode = cloneNode->next;
    }
    return clone;
}
```

clone LL with random ptrs

```cpp
Node* sublinkedList(Node* l1, Node* l2) {
    l1 = reverse(l1);
    l2 = reverse(l2);

    Node *dummy = new Node(0);
    Node *prev = dummy;
    int carry = 0;
    while(l1 || l2){
        int diff = carry;
        if(l1){
            diff += l1->data;
            l1 = l1->next;
        }
        if(l2){
            diff -= l2->data;
            l2 = l2->next;
        }
        if(diff < 0){
            diff += 10;
            carry = -1;
        }
        else
            carry = 0;
        prev->next = new Node(diff);
        prev = prev->next;
    }
    return reverse(dummy->next);
}
```

sub 2 LL

```cpp
Node* addlinkedList(Node* l1, Node* l2) {
    l1 = reverse(l1);
    l2 = reverse(l2);

    Node *dummy = new Node(0);
    Node *prev = dummy;
    int carry = 0;
    while(l1 || l2){
        int sum = carry;
        if(l1){
            sum += l1->data;
            l1 = l1->next;
        }
        if(l2){
            sum += l2->data;
            l2 = l2->next;
        }
        carry = sum%10;
        if(sum >9){
            sum /= 10;
        }
        else
            carry = 0;
        prev->next = new Node(sum);
        prev = prev->next;
    }
    return reverse(dummy->next);
}
```

Add 2 LL

```cpp
* Let L1 = distance from head to starting point of cycle
*     L2 = distance from starting point of cycle to intersection ppoint of slow ans fast ptr
*
* Dist travelled by slow ptr = L1 + L2
* Dist travelled by fast ptr = L1 + L2 + nC    where C -> Length of cycle
*
*     2 * Speed of slow ptr = fast ptr
* =>  2 (L1 + L2) = L1 + L2 + nC
* =>  L1 + L2 = nC
* => L1 = nC - L2
* |
* Which basically means DIST FROM HEAD TO STARTING POINT = DIST FROM INTERSECTION POINT OF SLOW AND FAST TO STARTING PT OF CYCLE
Node* LoopedOrNot(Node **head){  //T=O(n) S=O(1)
    Node *slowptr = *head;
    Node *fastptr = *head;
    bool looped = false;
    while(fastptr && slowptr && fastptr->next){
        slowptr = slowptr->next;
        fastptr = fastptr->next->next;
        if(slowptr == fastptr){
            looped = true;
            break;
        }
    }
    if(looped){
        slowptr = *head;
        while(slowptr != fastptr){
            slowptr = slowptr->next;
            fastptr = fastptr->next;
        }
        return slowptr;
    }
    return NULL;
}
```

Starting point of a cycle

```cpp
Node *getPivot(Node *head, int pivotIdx){
    if (head == nullptr) return nullptr;

    Node *pivot = head;
    while (pivot && pivotIdx--){
        pivot = pivot->next;
    }

    return pivot;
}

vector<Node*> segPivot(Node *head, int pivotIdx){
    Node *pivot = getPivot(head, pivotIdx);

    Node *smaller = new Node(-1);A
    Node *prevS = smaller;

    Node *larger = new Node(-1);
    Node *prevL = larger;

    Node *curr = head;
    int idx = 0;
    while(curr){
        if (idx == pivotIdx); //Do nothing
        else if(curr->data <= pivot->data){
            prevS->next = curr;
            prevS = prevS->next;
        }
        else{
            prevL->next = curr;
            prevL = prevL->next;
        }
        curr = curr->next;
        idx++;
    }
    prevS->next = nullptr;
    pivot->next = nullptr;
    prevL->next = nullptr;

    vector<Node*> vec = {smaller->next, pivot, larger->next};
    return vec;
}

int length(Node *head){
    int count = 0;
    Node *curr = head;
    while(curr){
        curr = curr->next;
        count++;
    }
    return count;
}

vector <Node *> mergeSortedLists(vector <Node *> leftSorted, Node *pivot, vector <Node *> rightSorted){
    Node *head = nullptr, *tail = nullptr;
    if(leftSorted[0] && rightSorted[0]){
        leftSorted[1]->next = pivot;
        pivot->next = rightSorted[0];
        head = leftSorted[0];
        tail = rightSorted[1];
    }

    else if(leftSorted[0] != nullptr){
        leftSorted[1]->next = pivot;
        head = leftSorted[0];
        tail = pivot;
    }

    else if(rightSorted[0] != nullptr){
        pivot->next = rightSorted[0];
        head = pivot;
        tail = rightSorted[1];
    }

    else{
        head = tail = pivot;
    }

    vector <Node *> vec = {head, tail};
    return vec;
}

vector<Node*> quickSort(Node *head){
    if (!head || !head->next){
        vector<Node*> vec = {head, head};
        return vec;
    }
    int len = length(head);
    int pivotIdx = len/2;

    vector<Node*> segratedLists = segPivot(head, pivotIdx);

    vector<Node *> leftSorted = quickSort(segratedLists[0]);
    vector<Node *> rightSorted = quickSort(segratedLists[2]);
    return mergeSortedLists(leftSorted, segratedLists[1], rightSorted);
}

Node *QuickSort(Node *head){
    vector <Node *> ans = quickSort(head);
    return ans[0];
}
```

QuickSort