DSA Mahaan H Placements ki Jaan H

TOPIC – Linklist

Q1)Middle of the Linklist

```
    while (fast != NULL && fast->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;
    }

    return slow;
```

Q2) Delete Middle of LL

```
class Solution {
public:
    ListNode* deleteMiddle(ListNode* head) {
        if(head== NULL || head -> next ==NULL){
            return NULL;
        }
        struct ListNode* slow = head;
        struct ListNode* fast = head;
        fast = fast ->next->next;
        while(fast!=NULL && fast->next!=NULL){
            slow = slow->next;
            fast = fast->next->next;
        }
        struct ListNode* middle = slow->next;
        slow->next=slow->next->next;
        delete middle;
        return head;
    }
};
```

Q3)Detect Loop in Linklist

```
  while(fast != NULL && fast->next!=NULL){
       slow = slow->next;
       fast = fast->next->next;

  if(slow == fast){
       return true;
  }
  }
  return false;
```

Q4) Linklist Loop Length

```
int findlength(Node* slow,Node* fast){
    int count=1;
    fast=fast->next;
    while(slow!=fast){
        count++;
        fast=fast->next;
    }
    return count;
}

 // Function to find the length of a loop in the linked list.

 int countNodesinLoop(Node *head) {
     // Code here
      Node* slow=head;
    Node* fast=head;
    while(fast!=NULL && fast->next!=NULL){
        slow=slow->next;
        fast=fast->next->next;
        if(slow==fast) return findlength(slow,fast);
    }
    return 0;
```

Q5) Linklist Cycle Starting Point

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

    ListNode *slow = head;
    ListNode *fast = head;

    // Step 1: Detect cycle
    while (fast != NULL && fast->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;

        if (slow == fast)
            break;
    }

    // No cycle
    if (fast == NULL || fast->next == NULL)
        return NULL;

    // Step 2: Find starting node of cycle
    slow = head;
    while (slow != fast) {
        slow = slow->next;
        fast = fast->next;
    }

    return slow;
}
```

Q6) Intersction Point of Y LL

```
ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
   if(headA == NULL || headB == NULL) return NULL;

   struct ListNode* t1 = headA;
   struct ListNode* t2 = headB;
   while(t1 !=t2){
    t1= t1->next;
    t2=t2->next;
    if(t1==t2) return t1;
    if(t1==NULL)t1=headB;
    if(t2==NULL)t2=headA;
   }
   return t1;

}
```

Q7)Reverse a LL

```
struct ListNode* reverseList(struct ListNode* head) {
       struct ListNode* prev = NULL;
        struct ListNode* cu
        while (curr != NULL
           struct ListNode*
            curr->next = pr
            prev = curr;
            curr = nextNode;
        }
        return prev;
    }
```

RECURSIVE APPROACH

```
if(head==NULL || head->next ==NULL) return head;
ListNode* newHead = reverseList(head->next);
ListNode* front = head->next;
front ->next =head;
head->next =NULL;
return newHead;
```

Q8) LL is Palindrome or not

```c
struct ListNode* reverseList(struct ListNode* head) {
      struct ListNode* prev = NULL;
       struct ListNode* curr = head;
       while (curr != NULL) {
          struct ListNode* nextNode = curr->next;
           curr->next = prev;
           prev = curr;
           curr = nextNode;
       }
       return prev;
   }

   bool isPalindrome(struct ListNode* head) {
       if (!head || !head->next) return true;

       // Step 1: Find the middle (slow = mid)
       struct ListNode* slow = head;
       struct ListNode* fast = head;
       while (fast->next && fast->next->next) {
           slow = slow->next;
           fast = fast->next->next;
       }

       // Step 2: Reverse the second half
       slow->next = reverseList(slow->next);
      struct ListNode* second = slow->next;

       // Step 3: Compare first half and reversed second half
       struct ListNode* first = head;
       while (second != NULL) {
           if (first->val != second->val) return false;
           first = first->next;
           second = second->next;
       }
```

Q9) Add 1 to a No. Represent by LL

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

class Solution {
  public:
    Node* addOne(Node* head) {
        // Your Code here
        // return head of list after adding one
head = reverse(head);  // Reverse the list

    Node* curr = head;
    int carry = 1;

    while (curr && carry) {
        int sum = curr->data + carry;
        curr->data = sum % 10;
        carry = sum / 10;

        if (!curr->next && carry) {
            curr->next = new Node(0); // Add new node if needed
        }

        curr = curr->next;
    }

    return reverse(head); }
};
```

Q10) ADD 2 No.’s In LL

```cpp
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        ListNode* t1 = l1;
        ListNode* t2 = l2;
        ListNode* dummyHead = new ListNode(-1);  // ✅ Correct cons
        ListNode* curr = dummyHead;
        int carry = 0;

        while (t1 != nullptr || t2 != nullptr) {
            int sum = carry;
            if (t1) {
                sum += t1->val;
                t1 = t1->next;
            }
            if (t2) {
                sum += t2->val;
                t2 = t2->next;
            }

            carry = sum / 10;
            ListNode* newNode = new ListNode(sum % 10);
            curr->next = newNode;
            curr = curr->next;
        }

        if (carry > 0) {
            curr->next = new ListNode(carry);
        }

        return dummyHead->next; // skip dummy node
```

Q11) Sort a LL of 0's,1's and 2's

```cpp
class Solution {
  public:
    Node* segregate(Node* head) {
        // code here
        if(head==NULL || head->next== NULL) return head;
        struct Node* dummyzero= new Node(-1);
        struct Node* dummyone= new Node(-1);
        struct Node* dummytwo= new Node(-1);
        struct Node* zero = dummyzero;
        struct Node* one = dummyone;
        struct Node* two = dummytwo;
        struct Node* temp = head;
        while(temp !=NULL){
            if(temp->data == 0){
                zero->next = temp;
                zero=temp;
            }
            else if(temp->data == 1){
                one->next = temp;
                one=temp;
            }
             else{
                two->next = temp;
                two=temp;
            }
            temp=temp->next;
        }
        zero->next=(dummyone->next)?dummyone->next : dummytwo->next;
        one->next=dummytwo->next;
        two->next = NULL;
        return dummyzero->next;
        delete dummyzero,dummyone,dummytwo;

    }
};
```

Q12) Delete Nth Node From last

```
    ListNode* fast = head;
    while(n--){
        fast = fast->next;
    }
      if(fast==NULL){
        return head->next;
    }
    ListNode* slow = head;
    while(fast->next!=NULL){
        slow=slow->next;
        fast = fast->next;
    }

    ListNode* delnode=slow->next;
    slow->next=slow->next->next;
    delete delnode;
    return head;
}
```

Q13) Odd and even place node seperate in LL

```
if(head==NULL || head->next == NULL)return head;
ListNode* odd = head;
ListNode* even = head->next;
ListNode* evenhead = even;
while(even !=NULL && even->next!= NULL){
    odd->next = odd->next->next;
    even ->next = even->next->next;

    odd=odd->next;
    even=even->next;

}
odd->next = evenhead;
return head;
```

Q14) INSERTION PROBLEMS

At front

```
struct Node* newnode = new Node(x);
newnode->next = head;
return newnode;
```

At End

```
Node* insertAtEnd(Node* head, int x) {
    Node* newnode = new Node(x);

    if (head == NULL) {
        return newnode;
    }
    Node* temp = head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newnode;
    return head;
}
```

AT Specific Position

```
Node* insertPos(Node* head, int pos, int val) {
    Node* newnode = new Node(val);

    if (pos == 1) {
        newnode->next = head;
        return newnode;
    }
    Node* temp = head;
    int count = 1;

    while (temp != NULL && count < pos - 1) {
        temp = temp->next;
        count++;
    }
    if (temp == NULL) {
        return head;
    }
    newnode->next = temp->next;
    temp->next = newnode;
    return head;
}
```

Q15)Delete particular node without access of head of LL

```
void deleteNode(ListNode* node) {

    node->val = node->next->val;
    ListNode* temp = node->next;
    node->next = node->next->next;
    delete temp;
}
```

Q16) Sort LL(Merge Sort)

```
ListNode* sortList(ListNode* head) {
    if (head == NULL || head->next == NULL)
     return head;

    ListNode* middle = findmiddle(head);
    ListNode* righthead= middle->next;
    middle->next=NULL;
    ListNode* lefthead=head;
    lefthead =sortList(lefthead);
    righthead=sortList(righthead);
    return mergetwoLists(lefthead , righthead);

}
```

```
ListNode* mergetwoLists(ListNode* list1 ,ListNode* list2 ){
   ListNode* dummyNode = new ListNode(-1);
    ListNode* temp = dummyNode;
    while (list1 != NULL && list2 != NULL) {
        if (list1->val < list2->val) {
        temp->next = list1;
        temp = list1;
        list1 = list1->next;
        } else {
        temp->next = list2;
        temp = list2;
        list2 = list2->next;
        }
    }
    if (list1)
    temp->next = list1;
    else
    temp->next = list2;

return dummyNode->next;
}
ListNode* findmiddle(ListNode* head){
    ListNode* slow= head;
    ListNode* fast = head->next;
    while (fast != NULL && fast->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;
    }

    return slow;
}
```

Q17) Rotate LL K Times

```cpp
class Solution {
public:
    ListNode* findnth(ListNode* temp,int k){
        int count = 1;
        while(temp->next!=NULL){
            if(count==k){return temp;}
            count++;
            temp= temp->next;
        }
        return temp;

    }
    ListNode* rotateRight(ListNode* head, int k) {
        if(head==NULL || k==0) return head;
        ListNode* tail = head;
        int n = 1;
        while(tail->next!=NULL){
            n++;
            tail=tail->next;
        }
        if(k%n==0)return head;
        tail->next = head;
        k=k%n;

        ListNode* lastnode= findnth(head,n-k);
        ListNode* newhead= lastnode->next;
        lastnode->next=NULL;
        return newhead;
    }
};
```

Q18) Reverse Node in K- Group

```
ListNode* getkthNode(ListNode* temp, int k) {
    while (temp != NULL && k > 1) {
        temp = temp->next;
        k--;
    }
    return temp;
}
ListNode* reverseKGroup(ListNode* head, int k) {
    if (head == NULL || k == 1) return head;
    ListNode* temp = head;
    ListNode* prevlast = NULL;

    while (temp != NULL) {
        ListNode* kthnode = getkthNode(temp, k);
        // If less than k nodes remain
        if (kthnode == NULL) {
            if (prevlast) prevlast->next = temp;
            break;
        }
        ListNode* nextnode = kthnode->next;
        kthnode->next = NULL;
        // Reverse current group
        reverseLL(temp);
        if (temp == head) {
            head = kthnode;
        } else {
            prevlast->next = kthnode;
        }
        prevlast = temp;
        temp = nextnode;
    }
    return head;
```

Q19) Remove duplicate from Sorted LL

```
ListNode* deleteDuplicates(ListNode* head) {
    ListNode* temp = head;
    while(temp!=NULL && temp->next!=NULL){
        ListNode* nextnode = temp->next;
        while(nextnode!=NULL && nextnode->val == temp->val){
            nextnode = nextnode->next;
        }
        temp->next = nextnode;
        temp = temp->next;
    }
    return head;
}
```

TOPIC = Doubly LinkLIST

Q20)Reverse DLL

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

Q21)Insertion at Position in DLL

```
Node *insertAtPos(Node *head, int p, int x) {
    Node* newnode = new Node(x);
    Node* temp = head;
    int count = 0;
    while (temp != NULL && count < p) {
        temp = temp->next;
        count++;
    }
    if (temp == NULL)
        return head;
    newnode->next = temp->next;
    if (temp->next != NULL)
        temp->next->prev = newnode;
    temp->next = newnode;
    newnode->prev = temp;
    return head;
}
```

Q22)Deletion at any pos in DLL

```
Node* delPos(Node* head, int x) {

    if (head == NULL)
        return NULL;
    if (x == 1) {
        Node* temp = head;
        head = head->next;
        if (head != NULL)
            head->prev = NULL;
        delete temp;
        return head;
    }

    Node* temp = head;
    int count = 1;
    while (temp != NULL && count < x) {
        temp = temp->next;
        count++;
    }
    if (temp == NULL)
        return head;
    if (temp->next != NULL)
        temp->next->prev = temp->prev;
    if (temp->prev != NULL)
        temp->prev->next = temp->next;
    delete temp;
    return head;
}
```

Q23) Clone a LL with Random Pointer

```
Node* copyRandomList(Node* head) {
  Node* temp = head;
  // Insert copynode in between
  while(temp!=NULL){
    Node* copynode = new Node(temp->val);
    copynode->next = temp->next;
    temp->next = copynode;
    temp= temp->next->next;
}
temp = head;
// connect random pointers
while(temp!=NULL){
    Node* copynode = temp->next;
    if(temp->random !=NULL) copynode->random = temp->random->next;
    else copynode->random = NULL;

    temp=temp->next->next;
}
//connecting next pointers
Node* dummynode = new Node(-1);
Node* res = dummynode;
temp = head;
while(temp!=NULL){
    res->next = temp->next;
    temp->next=temp->next->next;
    res = res->next;
    temp=temp->next;
}
return dummynode->next;
```

Q24)Delete all occurrences of a key in DLL

```
void deleteAllOccurOfX(struct Node** head, int x) {
    Node* temp = *head;

    while (temp != NULL) {
        if (temp->data == x) {

            // If node to be deleted is head
            if (temp == *head) {
                *head = temp->next;
            }
            Node* nextnode = temp->next;
            Node* prevnode = temp->prev;
            // Fix next node's prev
            if (nextnode) {
                nextnode->prev = prevnode;
            }
            // Fix previous node's next
            if (prevnode) {
                prevnode->next = nextnode;
            }
            delete temp;
            temp = nextnode;
        }
        else {
            temp = temp->next;
        }
    }
}
```

Q25) Remove duplicates from a sorted DLL

```
    // Your code here
    Node* temp = head;
    while(temp!=NULL && temp->next!=NULL){
        Node* nextnode = temp->next;
        while(nextnode!=NULL && nextnode->data == temp->data){
            nextnode=nextnode->next;
        }
        temp->next = nextnode ;
        if(nextnode) nextnode->prev = temp;
        temp=temp->next;
    }
    return head;
```

Q26)Find pairs with given sum in DLL

```
vector<pair<int, int>> findPairsWithGivenSum(Node *head, int target) {
    vector<pair<int, int>> ans;

    if (head == NULL) return ans;
    // Initialize left pointer
    Node* left = head;
    // Initialize right pointer at tail
    Node* right = head;
    while (right->next != NULL) {
        right = right->next;
    }
    // Two pointer approach
    while (left != right && right->next != left) {
        int sum = left->data + right->data;

        if (sum == target) {
            ans.push_back({left->data, right->data});
            left = left->next;
            right = right->prev;
        }
        else if (sum < target) {
            left = left->next;
        }
        else {
            right = right->prev;
        }
    }
    return ans;
}
```

Q27) **Flattening a Linked List**

GFG

```
class Solution {
  public:
    Node* merge(Node* list1,Node* list2){
        Node* dummynode = new Node(-1);
        Node* res = dummynode;
        while(list1!=NULL && list2!=NULL){
            if(list1->data < list2->data){
                res->bottom = list1;
                res = list1;
                list1=list1->bottom;
            }
            else{
               res->bottom = list2;
               res = list2;
               list2 = list2->bottom;
            }
            res->next = NULL;
            if(list1){
                res->bottom = list1;
            //    list1 = list1->next;  no need we just connect it bcq list will be sorted
            }
            else{
                res ->bottom = list2;
            }

        }
        return dummynode->bottom;
    }
    Node *flatten(Node *root) {
        // code here
        if(root==NULL || root->next==NULL){
            return root;
        }
        Node* merged_head=flatten(root->next);
        return merge(root,merged_head);

    }
};
```

Q28) Flatten a Multilevel DLL

```
public:
    Node* flatten(Node* head) {
        if (!head) return head;
        dfs(head);
        return head;
    }
private:
    // returns tail of the flattened list
    Node* dfs(Node* curr) {
        Node* temp = curr;
        Node* last = curr;
        while (temp) {
            Node* nextNode = temp->next;
            // if child exists → flatten it
            if (temp->child) {
                Node* childHead = temp->child;
                Node* childTail = dfs(childHead);
                // connect temp → child
                temp->next = childHead;
                childHead->prev = temp;
                // connect childTail → nextNode
                if (nextNode) {
                    childTail->next = nextNode;
                    nextNode->prev = childTail;
                }
                temp->child = NULL;   // VERY IMPORTANT
                last = childTail;
                temp = childTail;
            } else {last = temp; }
            temp = temp->next;
        }
        return last;
```

Q29) Merge K Sorted Lists

```cpp
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        if (!l1) return l2;
        if (!l2) return l1;

        if (l1->val < l2->val) {
            l1->next = mergeTwoLists(l1->next, l2);
            return l1;
        } else {
            l2->next = mergeTwoLists(l1, l2->next);
            return l2;
        }
    }

    ListNode* mergeKLists(vector<ListNode*>& lists) {
        if (lists.size() == 0) return NULL;

        ListNode* result = lists[0];

        for (int i = 1; i < lists.size(); i++) {
            result = mergeTwoLists(result, lists[i]);
        }

        return result;
    }
};
```