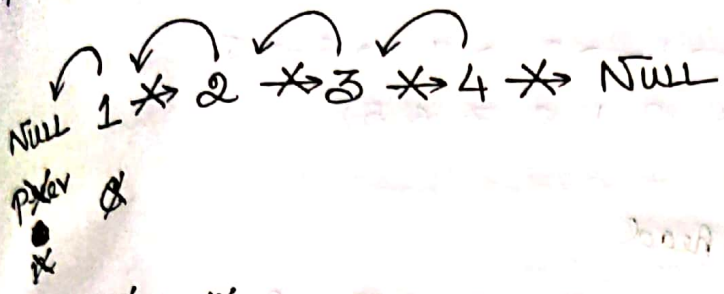
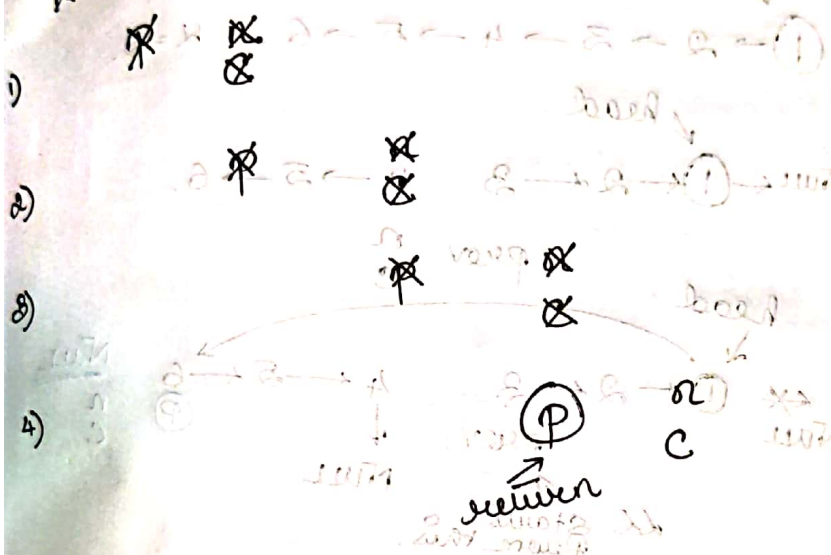
Linked List

LOVE BABBAR 450 DSA

1# Reverse a Linked List

```
class Node{
    int data;
    Node * next;
}
```





Note: n is written above c.

This is because in every iteration first n comes to c → next;

Then c is made equal to n.

Code

```
Node* reverse (Node* head){
    Node* current = head;
    Node * prev = NULL;
    Node * next = NULL;

    while (current != NULL){
        n = current → next;
        current → next = prev;
        prev = current;
        current = n;
    }

    return prev;
}
```

2) # Reverse Linked List in Groups of Size k **

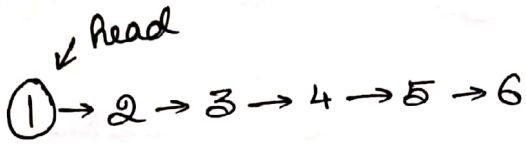
I/P:- 1→2→2→4→5→6→7→8

k = 4.

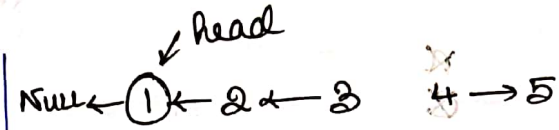
O/P:- 4→2→2→1→8→7→6→5.

(4 2 2 1 8 7 6 5).

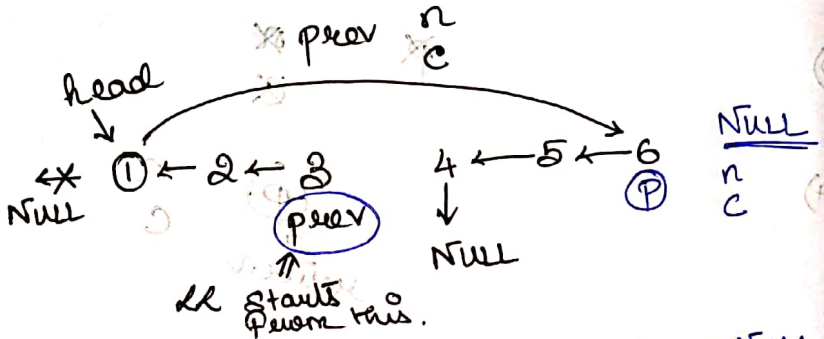
Algorithm →



After k=3 the posit^n. of ptrs



Recursively reverse next set and head→next set. to it.



O/P ← 3 → 2 → 1 → 6 → 5 → 4 → NULL

Code

```
Node* reverseKth(Node* head, int k){
    if (head == NULL || k == 1)
        return head;

    Node * current = head;
    Node * prev = NULL;
    Node * n = NULL;
    int count = 0;
    while(current != NULL && count < k){
        n = current → next;
        current → next = prev;
        prev = current;
        current = n;
        count ++;
    }
    if (current != NULL)
        head → next = reverseK(current, k);
    return prev;
}
```

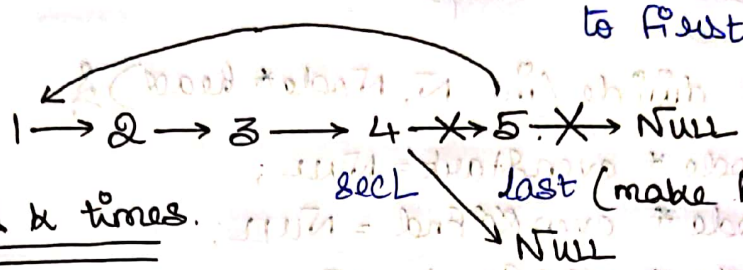
8) ** # Rotate Linked List by k positions Right

I/P: $1 \to 2 \to 3 \to 4 \to 5$; $k = 2$

O/P: $4 \to 5 \to 1 \to 2 \to 3$.

If $k=1$;
Code → Move Last Element to First.

algorithm



Do this k times.

Code

```
Node * rotateRight (Node* head, int k) {
    if (head == NULL)
        return head;
    int length=0;
    Node * temp = head;
    while(temp != NULL) {
        length++
        temp = temp → next;
    }
    Node *last = head, *secLast = NULL;
    k = k% length;
    for(int i=0; i< k; i++){
        while (last → next != NULL) {
            secLast = last;
            last = last → next;
        }
        secLast → next = NULL;
        last → next = head;
        head = last;
    }
    return head;
}
```

4)# Segregate Even and Odd Nodes in a LL

I/P : 17 → 15 → 8 → 9 → 2 → 4 → 6 → NULL

O/P : 8 2 4 6 17 15 9.

Code (Self Explanatory).

```
Node* divide(int N, Node* head){
    Node* evenStart = NULL;
    Node* evenEnd = NULL;
    Node* oddStart = NULL;
    Node* oddEnd = NULL;
    Node* current = head;
    while(current != NULL){
        int val = current->data;
        if(val%2==0){
            if(evenStart==NULL){   //first Even Node
                evenStart = current;
                evenEnd = evenStart;
            }else{
                evenEnd->next = current;
                evenEnd = current;
            }else{
                if(oddStart==NULL){   //first Odd Node
                    oddStart = current;
                    oddEnd = oddStart;
                }else{
                    oddEnd->next = current;
                    oddEnd = current
                }
            }
        curr = current->next;
    }
```

```
if (oddStart == NULL || evenStart == NULL)
    return head;

Node * head_ref = evenStart
evenEnd -> next = oddStart;
oddEnd -> next = NULL;

return head_ref;
```

#5) Remove Duplicates in a Sorted Linked List

I/P: 1 → 2 → 2 → 4 → 5 → NULL
O/P: 1 → 2 → 4 → 5 → NULL

Algorithm

① Compare with the next; if equal then duplicate present.
② Change ptr current → next = current → next → next

Code

```
Node* removeDuplicates (Node *head) {
    Node *current = head;
    if (head == NULL)
        return NULL;

    while (current → next != NULL) {
        if (current → data == current → next → data)
            current → next = current → next → next;
        else
            current = current → next
    }
    return head;
}
```

#6) Remove Duplicates in an Unsorted Linked List

Algorithm

① Use a hashmap to check whether an element is already present.

② Maintain 2 pointers (prev, current)

So prev → next = current → next;

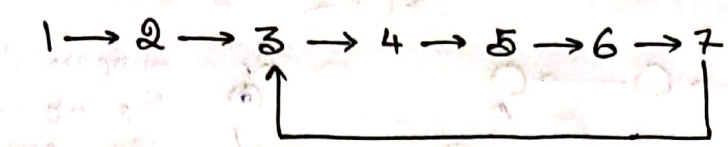
if duplicate.

I/P	⟶	5 → 1 → 2 → 2 → 4 → NULL
O/P	⟶	5 → 1 → 2 → 4 → NULL

Code

```
Node* removeDuplicates (Node* head){
    unordered_map<int, bool> hash;
    Node *current = head;
    hash[current → data] = true; //Present.
    Node *prev = current;
    current = current → next;
    while(current != NULL){
        if(hash[current → data]){          //Already present
            prev → next = current → next;  So duplicate
            free(current);                 Remove
        }else{
            hash[current → data]=true;
            prev = current;
        }
        current = prev → next;
    }
    return head;
}
```

#7) Detect a Cycle in a Linked List

I/P: 

① The slow_ptr will make 1 loop always and meet fast_ptr last than

② Fast_ptr will make 2 loops by then.

O/P: True.

Algorithm Floyd's Cycle Detection algo.

① Maintain 2 ptrs slow & fast.

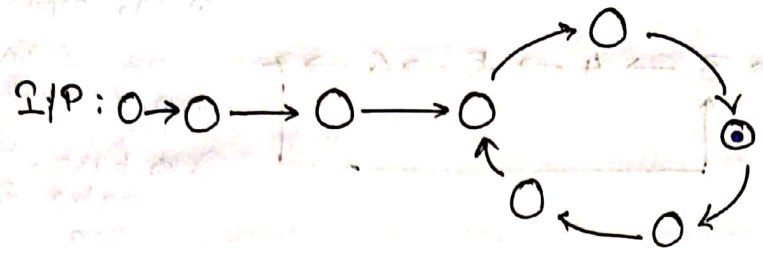
② Increment slow by 1.
Increment fast by 2.

③ If the ptrs ever meet again. Cycle is present.

Code

```
bool detectLoop(Node* head){
    Node* slowPtr = head;
    Node* fastPtr = head;
    while(fastPtr != NULL && fastPtr->next != NULL){
        slowPtr = slowPtr->next;
        fastPtr = fastPtr->next->next;
        if(slowPtr == fastPtr){ // they meet
            return true;        // Cycle detected
        }
    }
    return false;
}
```

#8) Detect the Starting pt. of a Cycle



Suppose this is the pt where the slow-ptr and fast-ptr meets.

① After detecting loop.
Make slow-ptr = head.

② Now increment both slow-ptr and fast-ptr by 1.

It is mathematically proven that the node where fast-ptr and slow-ptr meets again is the starting pt.

Code

```
Node* detectStartofCycle(Node* head){
    if (head == NULL)
        return head;
    Node* slowPtr = head;
    Node* fastPtr = head;
    bool cycleDetected = false;
    while(fastPtr != NULL && fastPtr->next != NULL){
        slowPtr = slowPtr->next;
        fastPtr = fastPtr->next;
        if (slowPtr == fastPtr){
            cycleDetected == true;
            break;
        }
    }
    if (cycleDetected){
        slowptr = head;
        while (slowPtr != fastPtr){
            slowPtr = slowPtr->next;
            fastPtr = fastPtr->next;
        }
        return slowPtr;  // Position of Starting Pt. of Loop;
    }else return NULL;
}
```

#a) Delete Loop in a Linked List.

I/P: 1 → 2 → 3 → 4 → 5 (5 → 2)

O/P:- 1 → 2 → 3 → 4 → 5 → NULL

Two cases
① Starting pt. of loop is head
② Starting pt. of loop is not head.

So if need to keep track of element b4 start of loop (5)
∴ cond → fastPtr → next.

Code

```
void removeLoop(Node* head){
    Node* slowPtr = head;
    Node * fastPtr = head;
    bool cycleDetected = false;
    while (fastPtr != NULL && fastPtr->next != NULL){
        slowPtr = slowPtr->next;
        fastPtr = fastPtr->next->next;
        if(slowPtr == fastPtr){
            cycleDetected = true;
            break;
        }
    }
    if (cycleDetected){
        if (slowPtr == head){ // Starting pt. of loop is head
            while(fastPtr->next != slowPtr)
                fastPtr = fastPtr->next;
            fastPtr->next = NULL;
        }
        else if(slowPtr == fastPtr){
            slowPtr = head;
            while(slowPtr->next != fastPtr->next){
                slowPtr = slowPtr->next;
                fastPtr = fastPtr->next;
            }
            fastPtr->next = NULL;
        }
    }
}
```

// Take another loop across.

// Make sure if we get element b4.

#10) Add "1" to a number represented by a LL.

I/P: 9 → 9 → 9

O/P: 1 → 0 → 0 → 0

I/P: 3 → 4 → 5

O/P: 3 → 4 → 6

Algorithm :- Reverse the number and perform (Then we will be directly performing on last digit)

Edge Case

9 → 9 → 9 ⟷ 1 ← 0 ← 0 ← 0 (Add 0 to the end make it).

(Reverse) 1 → 0 → 0 → 0 reverse

Code <Self Explanatory>

```
Node *addOne (Node * head){
    head = reverse (head);
    Node* current = head;

    while(current != NULL){
      if (current → next == NULL && current → data == 9){
        current → data = 1;          //Case of all 9's.
        Node* add = new Node(0);     //The loop has come this far indicates the above case.
        add → next = head;
        head = add;
        current = current → next;
      }
      else if (current → data == 9){
        current → data = 0;
        current = current → next;
      }
      else if (current → data != 9){   //Since we need to add "1". If digit is not 9 no carry generated hence break.
        current → data ++;
        current = current → next;
        break;
      }
    }
    return reverse (head);
}
```

11) Add two numbers represented by LL.

Similar to the prev. question — Reverse and add.

[If one number has more digits]
[Take 0 for the other]

sum = carry + first + second
carry = if sum >= 10 ; carry = 1
if sum >= 10 ; sum %= 10 ; (to get last digit)

i.e., check if the number is NULL take 0; otherwise data.

Code

```
Node* addTwo (Node* first, Node* second) {
    first = reverse (first);
    second = reverse (second);

    int sum = 0, carry = 0;
    Node * current = NULL;
    Node * head_ref = NULL;
    while(first != NULL || second != NULL) {
        sum = carry + (first ? first->data : 0)
                    + (second ? second -> data : 0);
        carry = (sum >= 10 ? 1 : 0);
        sum %= 10;
        Node* res = new Node(sum);
        if (head_ref == NULL)
            head_ref = res;
        else
            current -> next = res;
        current = res;

        if (first)
            first = first -> next;
        if (second)
            second = second -> next;
    }
    if (carry > 0) {
        Node * res = new Node (carry);
        current -> next = res;
        current = res;
    }
    return reverse (head_ref);
}
```

// If first take data or 0.

// For first node in the resultant.

#12 Intersection of two sorted Linked List

I/P: 1→2→3→4→6
2→4→6→8

O/P: 2→4→6

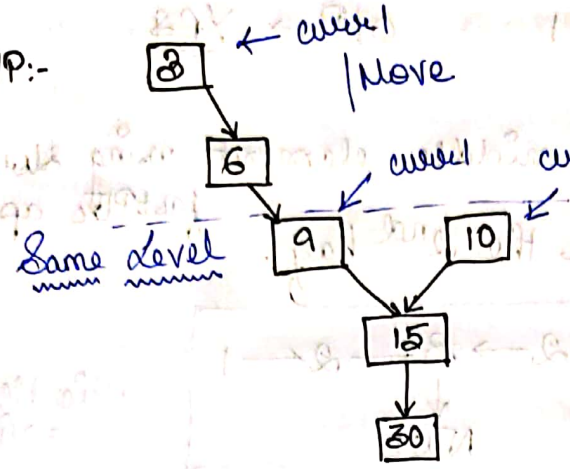
if (first→data) > (second→data)
move first ahead
• Same for second.

Code (Self Explanatory)

```
Node * findIntersection(Node* head1, Node* head2){
    Node* head_ref = NULL;      // for the resultant
    Node* current = NULL;       // Linked list

    Node* curr1 = head1;
    Node* curr2 = head2;

    while(curr1 != NULL && curr2 != NULL){

        if(curr1->data < curr2->data)
            curr1 = curr1->next;

        else if(curr2->data < curr1->data)
            curr2 = curr2->next;

        else if(curr1->data == curr2->data){
            Node* res = new Node(curr1->data);
            if(head_ref == NULL)    // for first node
                head_ref = res;
            else
                current->next = res;
            current = res;

            if(curr1)
                curr1 = curr1->next;
            if(curr2)
                curr2 = curr2->next;
        }
    }
    return head_ref;
}
```

#13) Intersection Point of two Linked List

I/P:-



O/P:- 15

len1: 5; len2: 3

Bring the ptr to the same level and then begin the iteration.

When ptrs are same that is intersection pt.

Code

```
int intersectPt(Node* head1, Node* head2){
    Node *curr1 = head1, *curr2 = head2;
    int len1 = 0, len2 = 0;
    while(curr1){
        len1++;
        curr1 = curr1->next;
    }
    while (curr2){
        len2++;
        curr2 = curr2->next;
    }
    curr1 = head1, curr2 = head2;   // Making the ptrs pt. to head again after calculating lengths
    int diff = abs(len1 - len2);
    if (len1 > len2){
        for(int i=0; i<diff; i++){
            curr1 = curr1->head;
    } else{
        for(int i=0; i<diff; i++)
            curr2 = curr2->head;
    }                                   // Making the ptr point to the same level
    while (curr1 != curr2){
        curr1 = curr1->next;
        curr2 = curr2->next;
    }
    if (curr1) return curr1->data;
    else return -1;
}
```

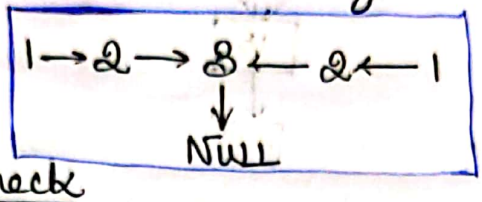
#14) Write a Program to check whether LL is Palindrome

I/P: 1→2→3→2→1 O/P:- YES

Algorithm

① Find the middle element using slowPtr fastPtr approach

② Reverse the 2nd half.



Middle → next = NULL;

③ Check

Code

```
bool isPalindrome(Node *head){
    Node *slowPtr = head, *fastPtr = head;
    while(fastPtr != NULL && fastPtr->next != NULL){
        slowPtr = slowPtr->next;
        fastPtr = fastPtr->next->next;
    }

    Node *prev = NULL, *next;
    Node *curr = slowPtr;
    while(curr != NULL){
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }

    Node *temp = head;
    while(prev != NULL){
        if(temp->data != prev->data)
            return false;
        temp = temp->next;
        prev = prev->next;
    }
    return true;
}
```

#15) Delete Nodes with Greater Values on the right

I/P:- 12 → 15 → 10 → 11 → 5 → 6 → 2 → 3

O/P: 15 11 6 3

If any node has a higher value on the right.

Algorithm :-

① Reverse and then start traversing

② 12 ← 15 ← 10 ← 11 ← 5 ← 6 ← 2 ← 3

③ Maintain maxR value. (Now 3)

While traversing if current node has a value <= maximum. Delete the node

If current node > max; update max.

Code

```
Node * compute (Node * head){
    head = reverse(head);
    Node *curr = head;
    int maxRight = curr → data;
    Node * prev = curr;
    curr = curr → next;
    while(curr != NULL){
        if (curr → data >= maxRight){
            maxRight = curr → data;   //update maxRight
            prev = curr;
            curr = curr → next;
        } else {
            prev → next = curr → next;
            free(curr);
            curr = prev → next;
        }
    }
    return reverse(head);
}
```

#16) Sort a linked list of 0s, 1s and 2s

Algorithm

① Maintain a count array of size (3) which stores counts of 0s 1s and 2s corresponding to the index;

② Now traverse the LL and set the values accordingly.

Code (Self-Explainatory)

```
Node *sort(Node* head){
    int count[3] = {0,0,0};
    Node *temp = head;
    while(temp != NULL){
        count[temp->data]++;
        temp = temp->next;
    }
    temp = head;
    int i=0;
    while(temp != NULL){
        if(count[i] == 0)
            i++;                    // Increment
        else{
            temp->data = i;
            count[i]--;
            temp = temp->next;
        }
    }
    return head;
}
```

#17) Merge 'k' sorted LLs (V.V. Important).

I/P:- [[1→8→4→6], [2→4→5], [7→9]]

O/P:- 1→2→3→4→4→5→6→7→9.

Algorithm

① Idea is to traverse the LLs and store the values in a min Heap.

② Then pop the values of the minheap to a new Linked list.

Code (Implementation of the above ideal algorithm)

```
Node* mergeKLL(vector<Node*> &lists){
    priority_queue<int, vector<int>, greater<int>> minH;
    int k = lists.size();
    for(int i=0; i<k; i++){
        Node* head = lists[i];
        while(head){
            minH.push(head->val);
            head = head->next;
        }
    }
    Node* head = NULL, *curr = NULL;
    while(!minH.empty()){          // while minHeap is not empty
        if(head==NULL){
            head = new Node(minH.top());
            minH.pop;
            curr = head;
        }else{
            Node *temp = new Node(minH.top());
            minH.pop();
            curr->next = temp;
            curr = temp;
        }
    }
    return head;
}
```

Do this only for the first Node of LL.

Remaining Nodes of the Linked List.

#18) Merge Sort For Linked Lists

Algorithm:
① Idea is to implement mergesort on the linked list.
② Divide and conquer from the middle.

Code

```
Node *mergeSort (Node *head){
    mergeSortUtil (&head);
    return head;
}

Void mergeSortUtil (Node **head){
    Node * curr=*head;
    Node *first, *second;
    if (curr== NULL || curr→next == NULL)
        return;
    FindMid(curr, &first, &second);
    mergeSortUtil(&first);
    mergeSortUtil (&second);
    *head= merge(first, second);
}

Void findMid(Node * curr, Node ** first, Node **second
    Node *fast = curr→next;
    Node * slow = curr;
    while(fast != NULL){
        fast = fast → next;
        if (Fast != NULL){
            fast = fast → next;
            slow = slow → next;
        }
    }
```

Slow Ptr pt to the mid element.

```
    *first = curr;
    *second = slow->next;
    slow->next = NULL;
}

Node *merge(Node *a, Node *b)
{
    if (a == NULL)
        return b;
    else if (b == NULL)
        return a;

    Node *ans = NULL;

    if (a->data <= b->data) {
        ans = a;
        ans->next = merge(a->next, b);
    } else {
        ans = b;
        ans->next = merge(a, b->next);
    }

    return answer;
}
```

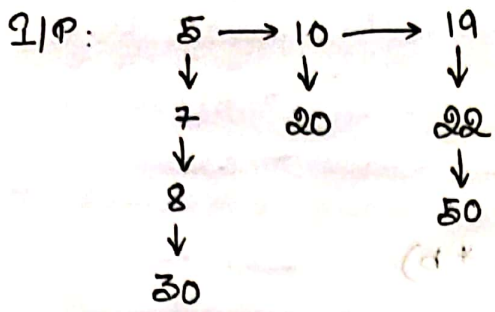
Divide array into 2 parts
first → mid
(mid+1) → last.

If one part of the LL is null return other.

Reduce size of a and call recursively

#19) Flatten a linked list.

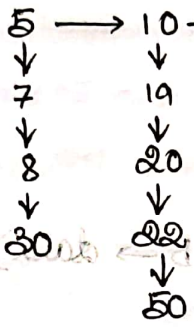
I/P:



O/P: 5 → 7 → 8 → 10 → 19 → 20 → 22 → 30 → 50

① Bottom ptr
② Next ptr

Algorithm:→ ① Start merging from the rightmost 2.



② Call recursively and merge.

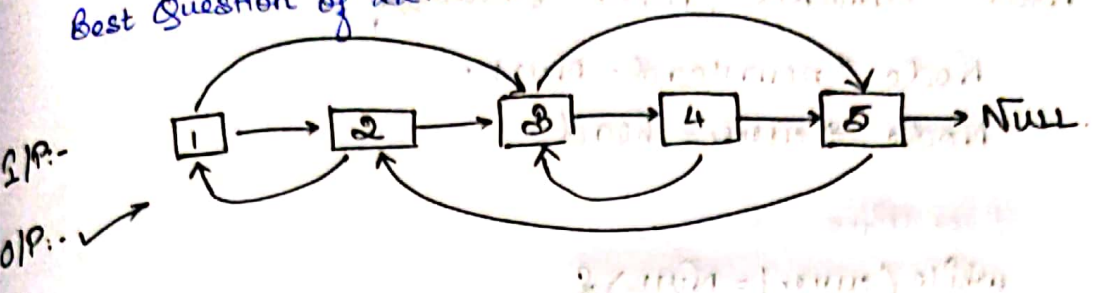
Code

```
Node *flatten(Node *root){
    if(root == NULL || root->next == NULL)
        return root;
    return merge(root, flatten(root->next));
}

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

    Node *res;
    if(first->data < second->data){
        res = first;
        res->bottom = merge(first->bottom, second);
    } else{
        res = second;
        res->bottom = merge(first, second->bottom);
    }

    res->next = NULL;
    return res;
}
```

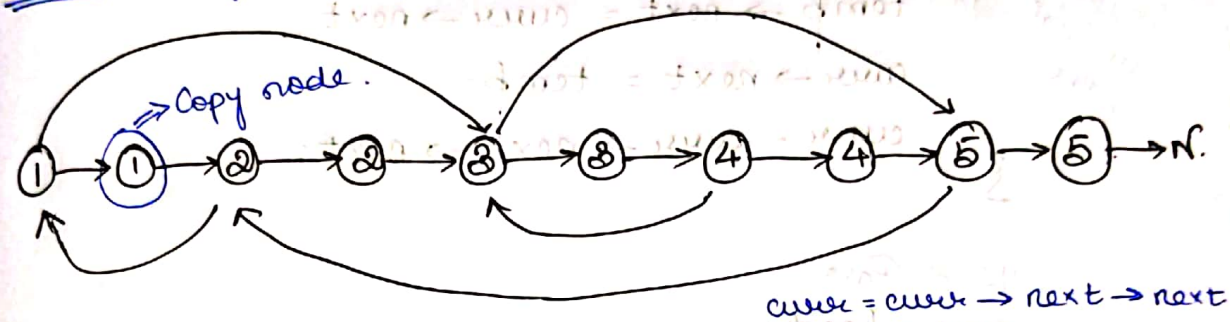
Merging happens downwards.

#80) Clone a Linked List with the next and random ptr

Best Question of LL



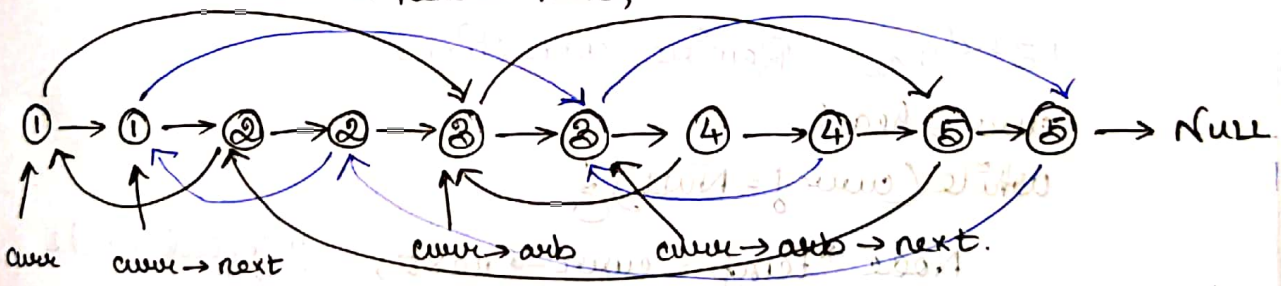
1st Pass (Copy nodes of Original)



curr = curr → next → next

2nd Pass (Fixing the random ptr of copied nodes).

```
if (curr → arb != NULL)   // Random ptr for original nodes exist.
    curr → next → arb = curr → arb → next;
curr = curr → next → next;
```



curr → next → arb = curr → arb → next.

Set (arb ptr of curr → next)

3rd Pass

Revise the connections (next connections) of the original and ~~copied~~ copied nodes.

Code

```
Node * cloneList(Node* &head){
    Node *newHead = NULL;
    Node *curr = head;

    // 1st Pass.
    while(curr != NULL){
        Node *temp = new Node(curr->data);
        temp->next = curr->next;
        curr->next = temp;
        curr = curr->next->next;
    }

    // 2nd Pass
    curr = head;
    while(curr != NULL){
        if(curr->arb != NULL)   // Random ptr exists
            curr->next->arb = curr->arb->next;
        curr = curr->next->next;
    }

    // 3rd Pass: Revise connections.
    curr = head;
    while(curr != NULL){
        Node *temp = curr->next;   // Copied node
        if(newHead == NULL)   // First node.
            newHead = temp;
        curr->next = curr->next->next;   // Original Node
        if(temp->next != NULL)
            temp->next = temp->next->next;
        curr = curr->next;
    }
    return newHead;
```

Copying original nodes.

If original node exists duplicate also exist.