Find intersection of Two Linked Lists

Problem Statement: Given the heads of two singly linked-lists **headA** and **headB**, return **the node at which the two lists intersect**. If the two linked lists have no intersection at all, return **null**.

Examples:

Example 1:

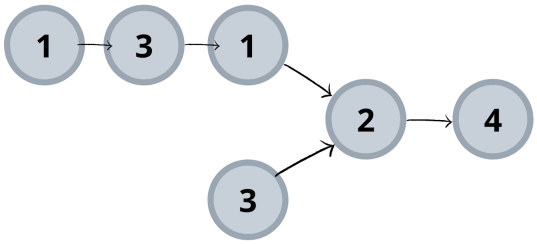
Input:

```
List 1 = [1,3,1,2,4], List 2 = [3,2,4]
```

Output:

```
2
```

Explanation: Here, both lists intersecting nodes start from node 2.



Example 2:

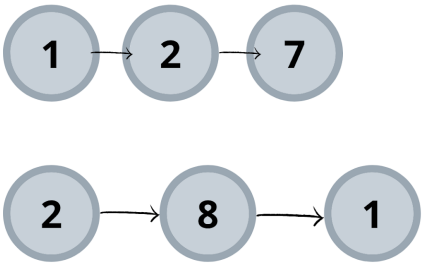
Input:

```
 List1 = [1,2,7], List 2 = [2,8,1]
```

Output:

```
 Null
```

Explanation: Here, both lists do not intersect and thus no intersection node is present.



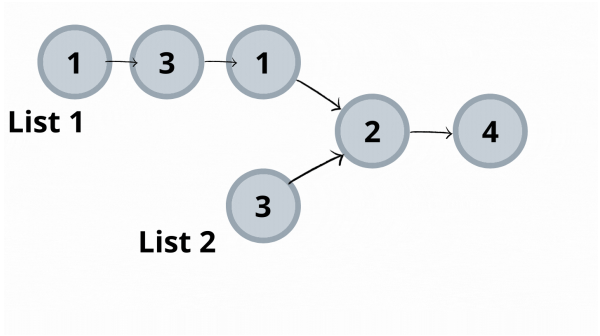
Solution 1: Brute-Force

Approach: We know intersection means a common attribute present between two entities. Here, we have linked lists as given entities.

What should be the common attribute for two linked lists?
If you believe a common attribute is a node's value, then think properly! If we take our example 1, there we can see both lists have nodes of value 3. But it is not the first intersection node. So what's the common attribute?
It is the node itself that is the common attribute. So, the process is as follows:-

- Keep any one of the list to check its node present in the other list. Here, we are choosing the second list for this task.
- Iterate through the other list. Here, it is the first one.
- Check if the both nodes are the same. If yes, we got our first intersection node.
- If not, continue iteration.
- If we did not find an intersection node and completed the entire iteration of the second list, then there is no intersection between the provided lists. Hence, return *null*.

Dry Run:



Code:

```
#include<iostream>
using namespace std;
//utility function to check presence of intersection
node* intersectionPresent(node* head1,node* head2) {
    while(head2 != NULL) {
        node* temp = head1;
        while(temp != NULL) {
            //if both nodes are same
            if(temp == head2) return head2;
            temp = temp->next;
        }
        head2 = head2->next;
    }
    //intersection is not present between the lists return null
    return NULL;
}
```

Output:
List1: 1->3->1->2->4
List2: 3->2->4
The intersection point is 2

Time Complexity: O(m*n)
Reason: For each node in list 2 entire lists 1 are iterated.

Space Complexity: O(1)
Reason: No extra space is used.

Solution 2: Hashing
Approach:
Can we improve brute-force time complexity? In brute force, we are basically performing “searching”. We can also perform searches by Hashing. Taking into consideration that hashing process takes O(1) time complexity. So the process is as follows:-

- Iterate through list 1 and hash its node address. Why? (Hint: depends on common attribute we are searching)
- Iterate through list 2 and search the hashed value in the hash table. If found, return node.

Code:

```
#include<bits/stdc++.h>
using namespace std;
class node {
    public:
        int num;
        node* next;
        node(int val) {
            num = val;
            next = NULL;
        }
};
//utility function to check presence of intersection
node* intersectionPresent(node* head1,node* head2) {
     unordered_set<node*> st;
    while(head1 != NULL) {
       st.insert(head1);
       head1 = head1->next;
    }
    while(head2 != NULL) {
        if(st.find(head2) != st.end()) return head2;
        head2 = head2->next;
    }
    return NULL;

}
```

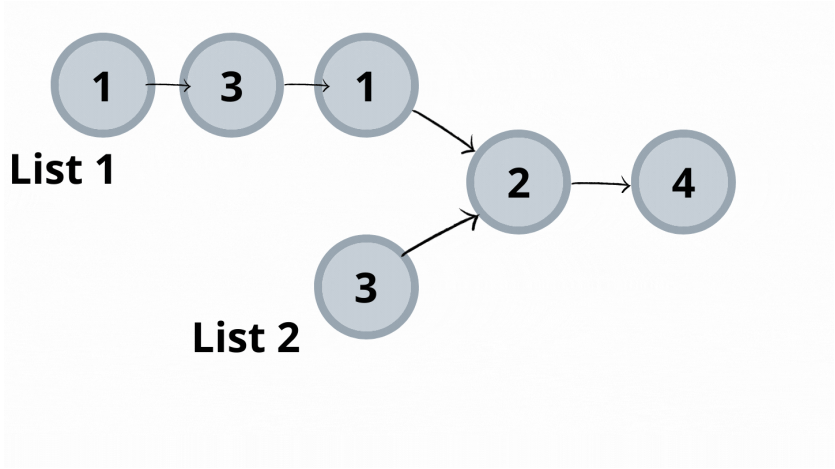
Output:
List1: 1->3->1->2->4
List2: 3->2->4
The intersection point is 2
Time Complexity: O(n+m)
Reason: Iterating through list 1 first takes O(n), then iterating through list 2 takes O(m).
Space Complexity: O(n)
Reason: Storing list 1 node addresses in unordered_set.

Solution 3: Difference of length
Approach:

We will reduce the search length. This can be done by searching the length of the shorter linked list. How? Let's see the process.

- Find length of both the lists.
- Find the positive difference of these lengths.
- Move the dummy pointer of the larger list by difference achieved. This makes our search length reduced to the smaller list length.
- Move both pointers, each pointing two lists, ahead simultaneously if both do not collide.

Dry Run:



Code:

```
#include<bits/stdc++.h>
using namespace std;

class node {
    public:
        int num;
        node* next;
        node(int val) {
            num = val;
            next = NULL;
        }
};
int getDifference(node* head1,node* head2) {
     int len1 = 0,len2 = 0;
        while(head1 != NULL || head2 != NULL) {
            if(head1 != NULL) {
                ++len1; head1 = head1->next;
            }
            if(head2 != NULL) {
                ++len2; head2 = head2->next;
            }

        }
        return len1-len2;//if difference is neg-> length of list2 > length of
list1 else vice-versa
}
```

```
//utility function to check presence of intersection
node* intersectionPresent(node* head1,node* head2) {
 int diff = getDifference(head1,head2);
        if(diff < 0)
            while(diff++ != 0) head2 = head2->next;
        else while(diff-- != 0) head1 = head1->next;
        while(head1 != NULL) {
            if(head1 == head2) return head1;
            head2 = head2->next;
            head1 = head1->next;
        }
        return head1;
}
```

OUTPUT:-

List1: 1->3->1->2->4
List2: 3->2->4
The intersection point is 2

Time Complexity:

O(2max(length of list1,length of list2))+O(abs(length of list1-length of list2))+O(min(length of list1,length of list2))

Reason: Finding the length of both lists takes max(length of list1, length of list2) because it is found simultaneously for both of them. Moving the head pointer ahead by a difference of them. The next one is for searching.

Space Complexity: O(1)

Reason: No extra space is used.

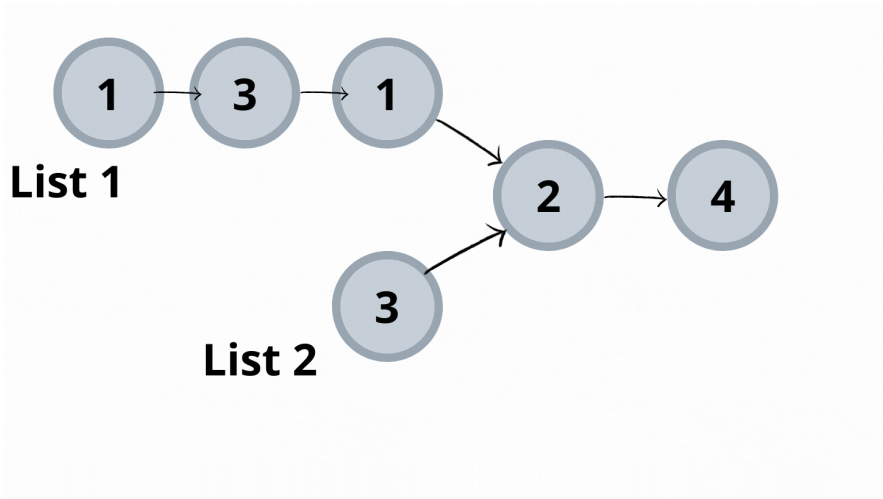
Solution 4: Optimised

Approach:

The difference of length method requires various steps to work on it. Using the same concept of difference of length, a different approach can be implemented. The process is as follows:-

- Take two dummy nodes for each list. Point each to the head of the lists.
- Iterate over them. If anyone becomes null, point them to the head of the opposite lists and continue iterating until they collide.

Dry Run:



Code:

```
//utility function to check presence of intersection
node* intersectionPresent(node* head1,node* head2) {
    node* d1 = head1;
    node* d2 = head2;

    while(d1 != d2) {
        d1 = d1 == NULL? head2:d1->next;
        d2 = d2 == NULL? head1:d2->next;
    }

    return d1;}
```

Output:
List1: 1->3->1->2->4
List2: 3->2->4
The intersection point is 2
Time Complexity: O(2*max(length of list1,length of list2))
Reason: Uses the same concept of difference of lengths of two lists.
Space Complexity: O(1)
Reason: No extra data structure is used

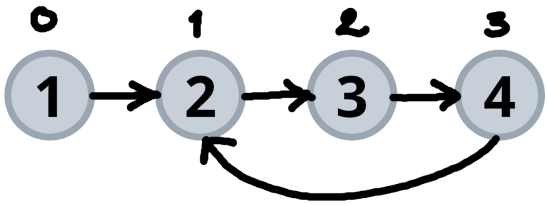
Detect a Cycle in a Linked List

In this article we will solve the most asked interview question: Detect a Cycle in a Linked List
Problem Statement: Given *head*, the head of a linked list, determine if the linked list has a cycle in it.
There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer.
Return *true* if there is a cycle in the linked list. Otherwise, return *false*.
Examples:

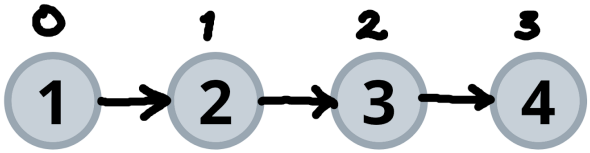
```
Example 1:
Input:
 Head = [1,2,3,4]
Output:
```

true

Explanation: Here, we can see that we can reach node at position 1 again by following the next pointer. Thus, we return true for this case.



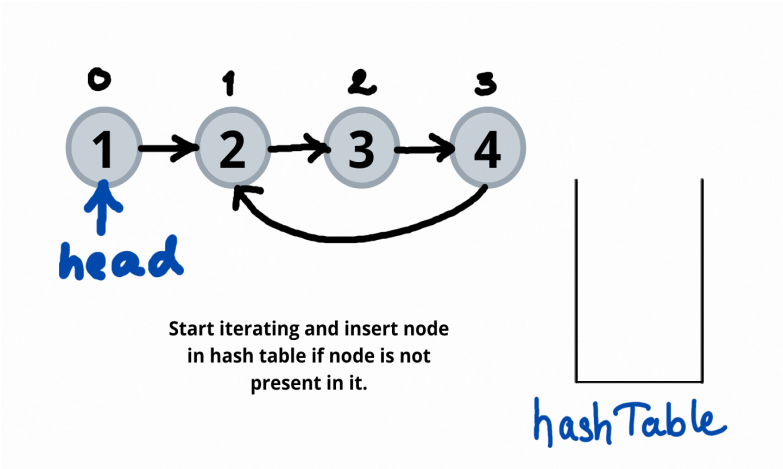
Example 2:
Input:
Head = [1,2,3,4]
Output:
false
Explanation: Neither of the nodes present in the given list can be visited again by following the next pointer. Hence, no loop exists. Thus, we return false.



Solution: Hashing
Approach:
We need to keep track of all the nodes we have visited till now so that once we visit the same node again we can say that a cycle is detected. The process is as follows:

- Use a hash table for storing nodes.
- Start iterating through the lists.
- If the current node is present in the hash table already, this indicates the cycle is present in the linked list and returns true.
- Else move insert the node in the hash table and move ahead.
- If we reach the end of the list, which is NULL, then we can say that the given list does not have a cycle in it and hence we return false.

Dry Run:



Code:

```
//utility function to detect cycle
bool cycleDetect(node* head) {
    unordered_set<node*> hashTable;
    while(head != NULL) {
        if(hashTable.find(head) != hashTable.end()) return true;
        hashTable.insert(head);
        head = head->next;
    }
    return false;
}
```

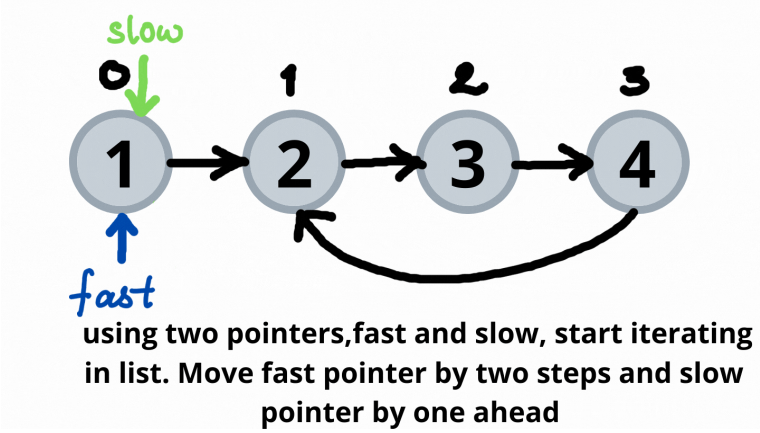
Output: Cycle detected
Time Complexity: O(N)
Reason: Entire list is iterated once.
Space Complexity: O(N)
Reason: All nodes present in the list are stored in a hash table.

Solution: Slow and Faster Pointer
Approach:
We will use two pointers with different steps forward. The process is as follows:-

- We will take two pointers, namely fast and slow. Fast pointer takes 2 steps ahead and slow pointer takes 2 points ahead.
- Iterate through the list until the fast pointer is equal to NULL. This is because NULL indicates that there is no cycle present in the given list.

- Cycle can be detected when fast and slow pointers collide.

Dry Run:



Code:

```
//utility function to detect cycle
bool cycleDetect(node* head) {
    if(head == NULL) return false;
    node* fast = head;
    node* slow = head;

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

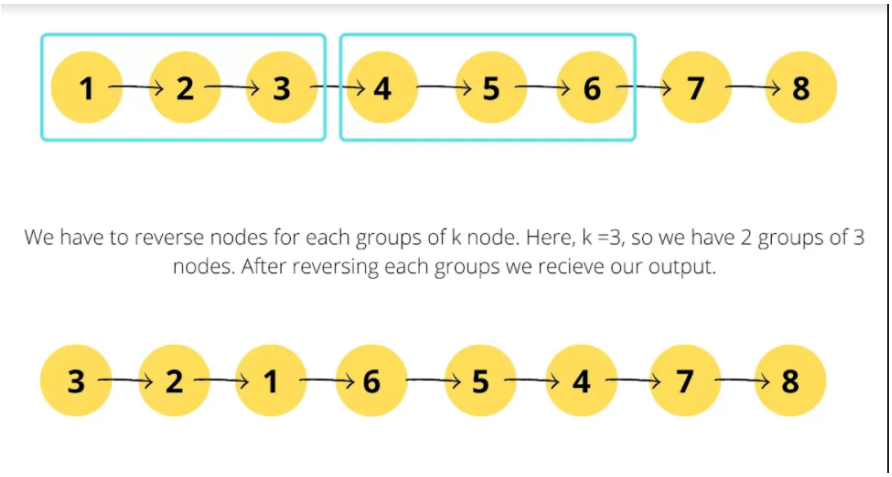
Output: Cycle detected
Time Complexity: O(N)
Reason: In the worst case, all the nodes of the list are visited.
Space Complexity: O(1)
Reason: No extra data structure is used.

Reverse Linked List in groups of Size K

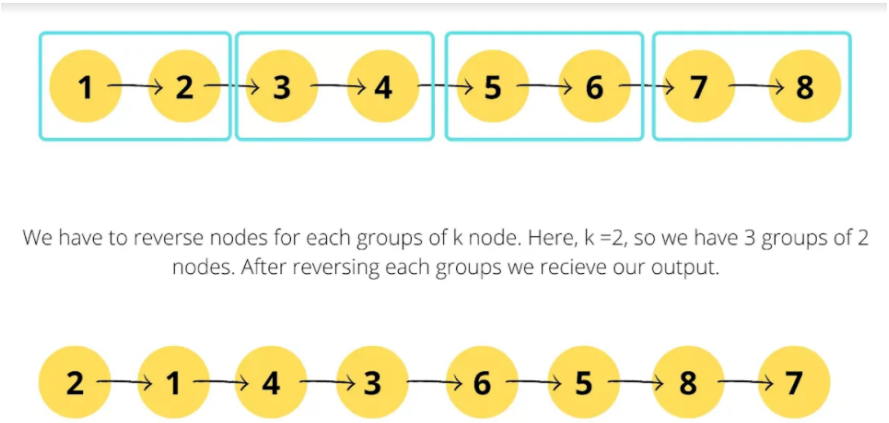
In this article we will solve a very popular question Reverse Linked List in groups of Size K.
Problem Statement: Given the head of a linked list, reverse the nodes of the list k at a time, and return *the modified list*. k is a positive integer and is less than or equal to the length of the linked list. If the number of nodes is not a multiple of k then left-out nodes, in the end, should remain as it is.
Examples:

Example 1:
Input:
```
 head = [1,2,3,4,5,6,7,8] k=3
```
Output:
```
 head = [3,2,1,6,5,4,7,8]
```

Explanation:



Example 2:
Input:
```
 head = [1,2,3,4,5,6,7,8] k=2
```
Output:
```
 head = [2,1,4,3,6,5,8,7]
```
Explanation:

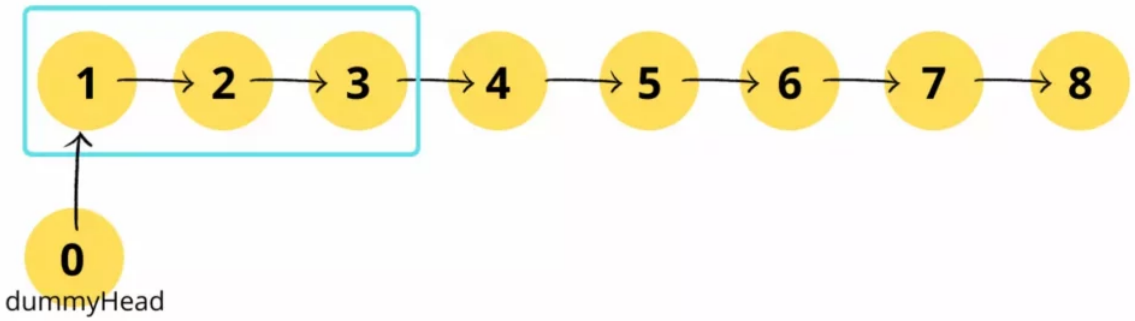


Solution:
Approach:

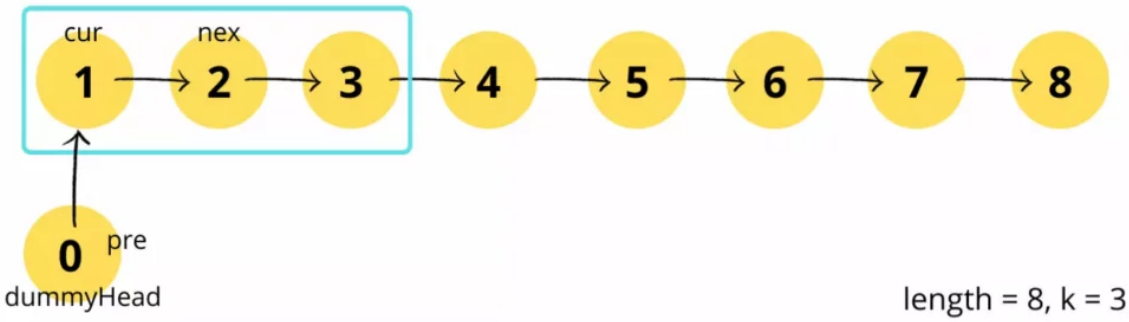
The following steps are needed to arrive at the desired output.

- Create a dummy node. Point next to this node to head of the linked list provided.
- Iterate through the given linked list to get the length of the list.
- Create three pointer pre,cur and nex to reverse each group. Why? If we observe output, we can see that we have to reverse each group, apart from modifying links of group.
- Iterate through the linked list until the length of list to be processed is greater than and equal to given k.
- For each iteration, point cur to pre->next and nex to nex->next.
- Start nested iteration for length of k.
- For each iteration, modify links as following steps:-
- cur->next = nex->next
- nex->next = pre->next
- pre->next = nex
- nex = cur->next
- Move pre to cur and reduce length by k.

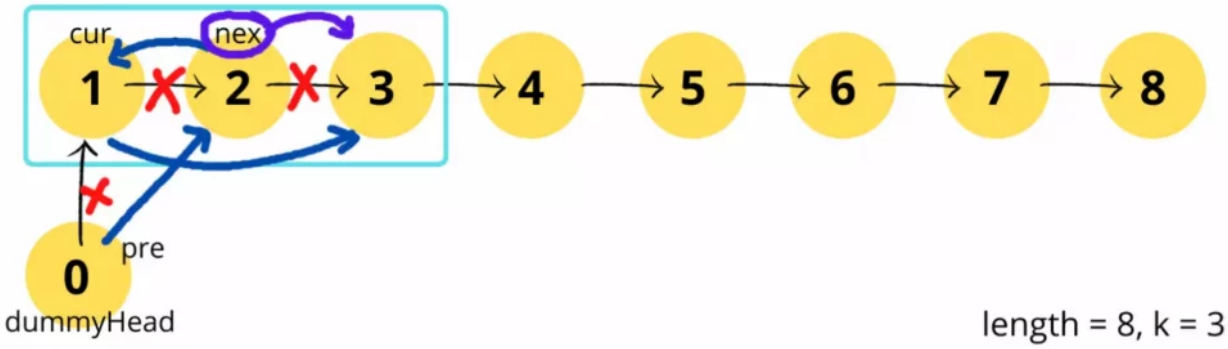
Dry Run:



We have to reverse nodes present in groups of length k. Let's start with first one. Create a dummy node with its next pointer pointing to head of the given list.

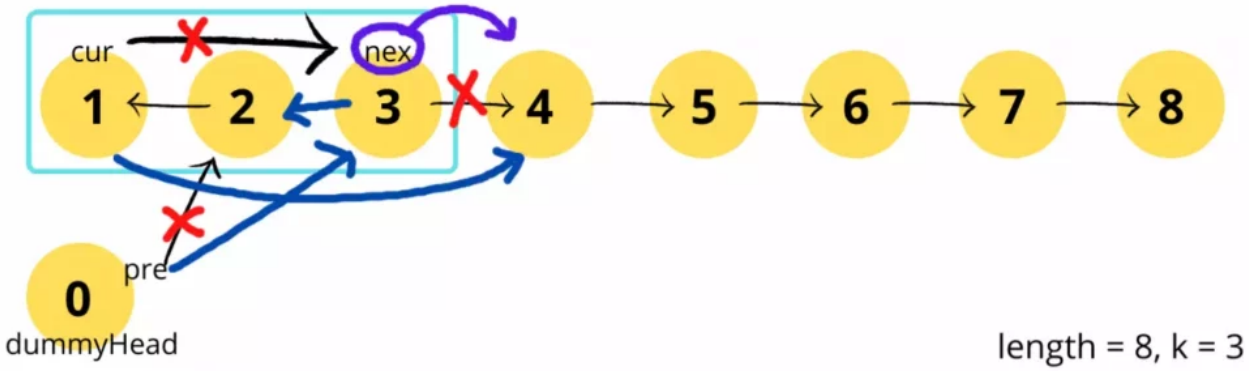


start iterating until length >= k. Initialise pre, cur and nex pointers.
pre = dummyHead, cur = pre->next, nex = cur->next



change links as per the following.

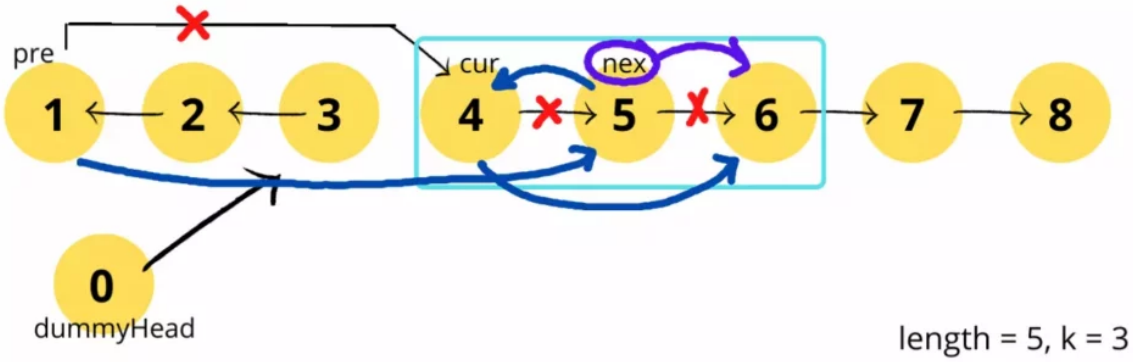
```
cur->next = nex->next;
nex->next = pre->next
pre->next = nex;
nex = cur->next;
```



Now, we have to point 3 -> 2. So, perform same steps again.

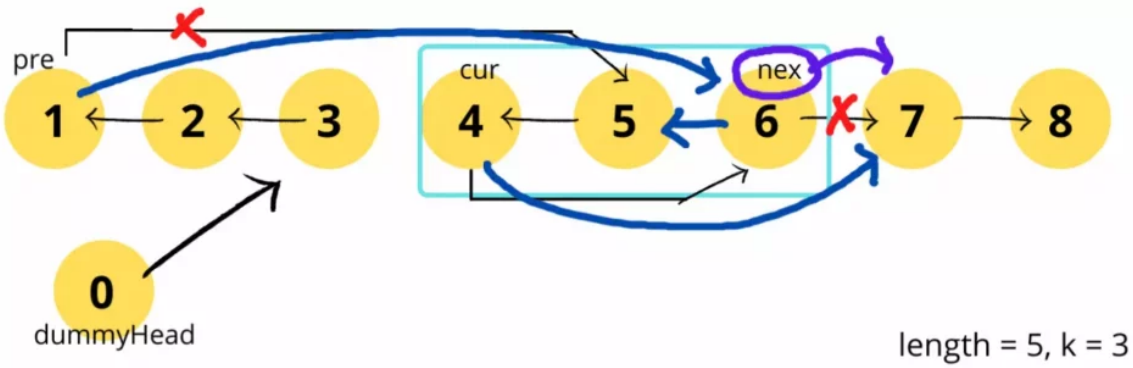
```
cur->next = nex->next;
nex->next = pre->next
pre->next = nex;
nex = cur->next;
```

d



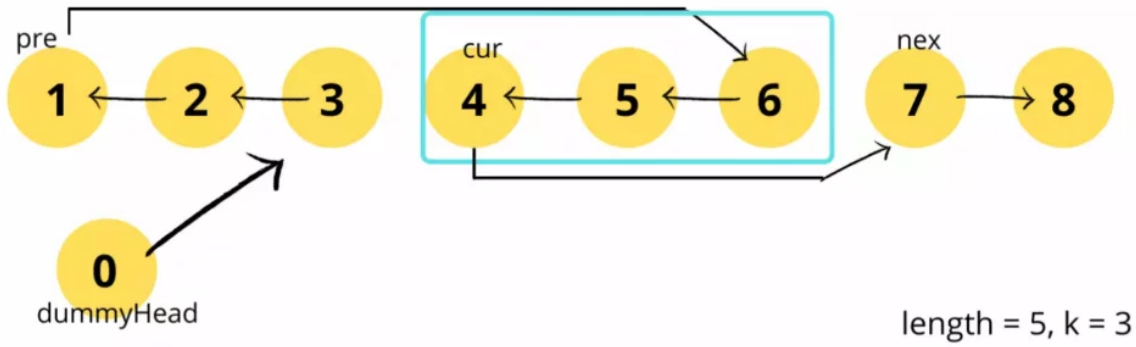
curr moves to pre->next and nex to nex->next. Again perform same set of questions.

```
cur->next = nex->next;
nex->next = pre->next
pre->next = nex;
nex = cur->next;
```

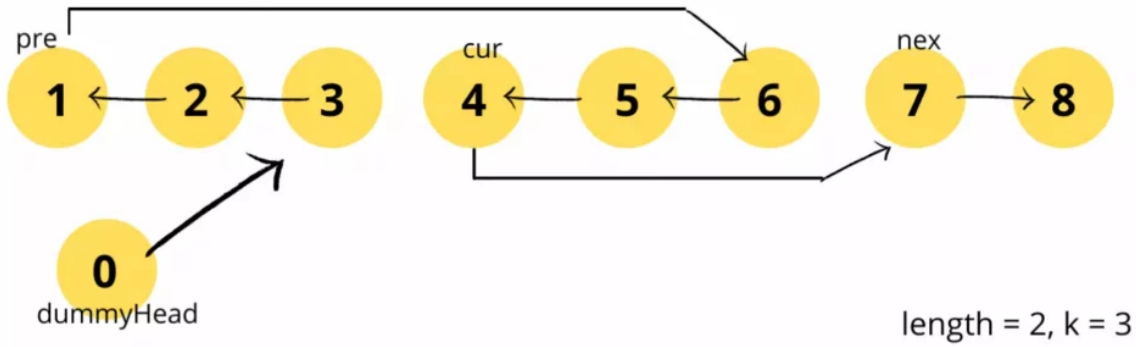


Now, we have to 6->5. Again perform same set of questions.

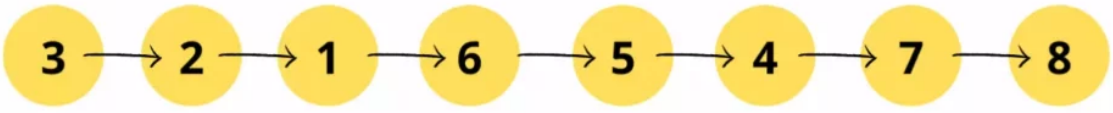
```
cur->next = nex->next;
nex->next = pre->next
pre->next = nex;
nex = cur->next;
```



We completed reversal of the 2nd group too. Move pre to cur and reduce length by k



length < k. Thus stop process. We recieved out output.



Code:

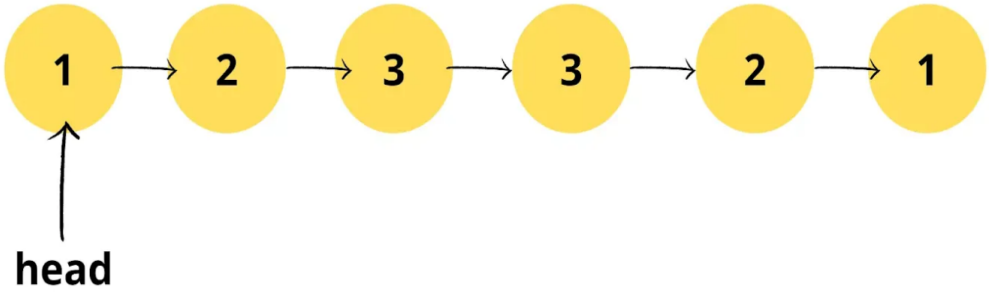
```
//utility function to reverse k nodes in the list
node* reverseKNodes(node* head,int k) {
    if(head == NULL||head->next == NULL) return head;

    int length = lengthOfLinkedList(head);

    node* dummyHead = new node(0);
    dummyHead->next = head;

    node* pre = dummyHead;
    node* cur;
    node* nex;

    while(length >= k) {
        cur = pre->next;
        nex = cur->next;
        for(int i=1;i<k;i++) {
            cur->next = nex->next;
            nex->next = pre->next;
            pre->next = nex;
            nex = cur->next;
        }
        pre = cur;
        length -= k;
    }
    return dummyHead->next;
}
```

Output:
Original Linked List: 1->2->3->4->5->6->7->8
After Reversal of k nodes: 3->2->1->6->5->4->7->8
Time Complexity: O(N)
Reason: Nested iteration with O((N/k)*k) which makes it equal to O(N).
Space Complexity: O(1)
Reason: No extra data structures are used for computation.

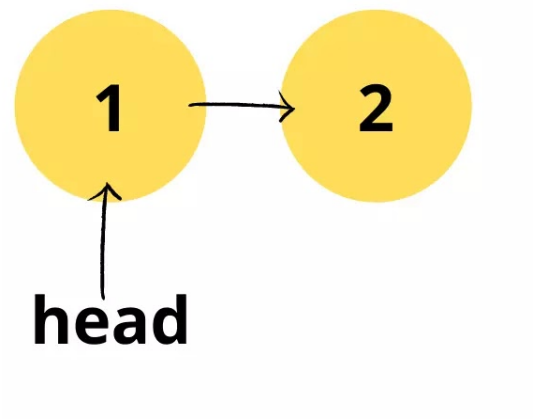
Check if given Linked List is Palindrome

Problem Statement: Given the head of a singly linked list, return true if it is a palindrome.
Examples:
Example 1:
Input: head = [1,2,3,3,2,1]
Output:
true

Explanation: If we read elements from left to right, we get [1,2,3,3,2,1]. When we read elements from right to left, we get [1,2,3,3,2,1]. Both ways list remains same and hence, the given linked list is palindrome.



Example 2:
Input:
head = [1,2]
Output:
false
Explanation: When we read elements from left to right, we get [1,2]. Reading from right to left, we get a list as [2,1]. Both are different and hence, the given linked list is not palindrome.



Solution: Using the extra data structure
Approach:
We can store elements in an array. Then check if the given array is a palindrome. How to check if an array is a palindrome?
Let's take a string, say "level" which is a palindrome. Let's observe a thing.

Letter	Position	Letter	Position
l	0	l	4
e	1	e	3
v	2	v	2

So we can see that each index letter is the same as (length-each index -1) letter.

The same logic required to check an array is a palindrome.
Following are the steps to this approach.

- Iterate through the given list to store it in an array.
- Iterate through the array.
- For each index in range of n/2 where n is the size of the array
- Check if the number in it is the same as the number in the n-index-1 of the array.

Code:

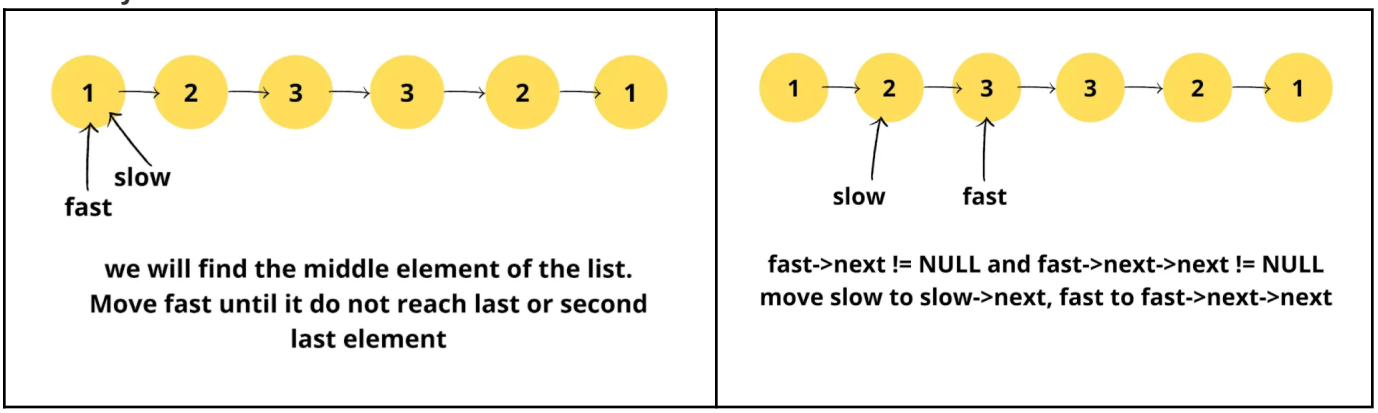
```
bool isPalindrome(node* head) {
    vector<int> arr;
    while(head != NULL) {
        arr.push_back(head->num);
        head = head->next;
    }
    for(int i=0;i<arr.size()/2;i++)
        if(arr[i] != arr[arr.size()-i-1]) return false;
    return true;
}
```

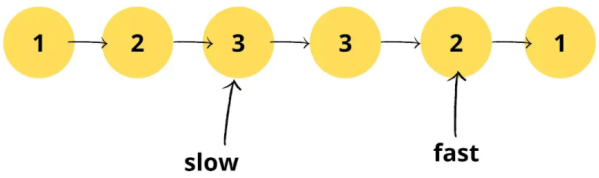
Output: True
Time Complexity: O(N)
Reason: Iterating through the list to store elements in the array.
Space Complexity: O(N)
Reason: Using an array to store list elements for further computations.

Solution 2: Optimized Solution
Approach:
Following are the steps to this approach:-

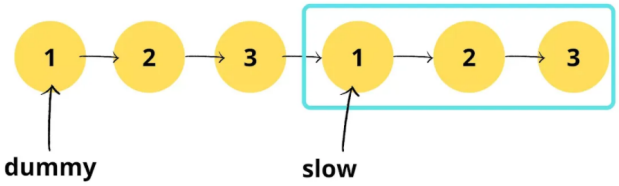
- Find the middle element of the linked list. Refer to this article to know the steps https://takeuforward.org/data-structure/find-middle-element-in-a-linked-list/
- Reverse linked list from next element of middle element. Refer to this article to know the steps https://takeuforward.org/data-structure/reverse-a-linked-list/
- Iterate through the new list until the middle element reaches the end of the list.
- Use a dummy node to check if the same element exists in the linked list from the middle element.

Dry Run:

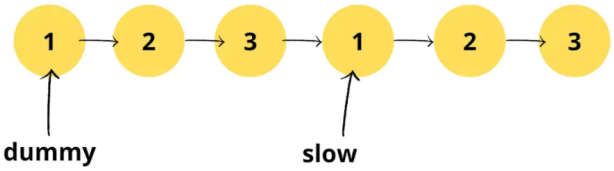




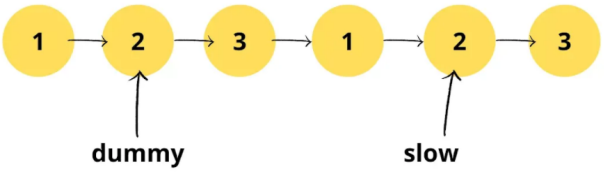
fast reached second last element. So we got oour middle element as 3. Now reverse the list from slow->next till end of the list.



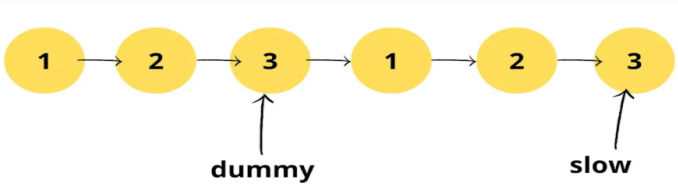
Reversed portion highlighted. Assign dummy node at the head of the list. Move slow pointer ahead. Start iterating until slow pointer reaches end of the list



dummy->num == slow->num, move dummy and slow ahead simultaneously



dummy->num == slow->num, move dummy and slow ahead simultaneously



dummy->num == slow->num, move dummy and slow ahead simultaneously. Slow reached end of the list and no element is different. Thus, list is palindrome.

Code:

```
node* reverse(node* ptr) {
    node* pre=NULL;
    node* nex=NULL;
    while(ptr!=NULL) {
        nex = ptr->next;
        ptr->next = pre;
        pre=ptr;
        ptr=nex;
    }
    return pre;
}
bool isPalindrome(node* head) {
    if(head==NULL||head->next==NULL) return true;

    node* slow = head;
    node* fast = head;

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

    slow->next = reverse(slow->next);
    slow = slow->next;
    node* dummy = head;

    while(slow!=NULL) {
        if(dummy->num != slow->num) return false;
        dummy = dummy->next;
        slow = slow->next;
    }
    return true;
}
```

Output: True
Time Complexity: O(N/2)+O(N/2)+O(N/2)
Reason: O(N/2) for finding the middle element, reversing the list from the middle element, and traversing again to find palindrome respectively.
Space Complexity: O(1)
Reason: No extra data structures are used.

Starting point of loop in a Linked List

In this article, we will learn how to solve the most asked coding interview question: "**Starting point of loop in a Linked List**"

Problem Statement: Given the head of a linked list, return *the node where the cycle begins. If there is no cycle, return* null.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that the tail's next pointer is connected to (0-indexed). It is -1 if there is no cycle. Note that pos is not passed as a parameter.

Do not modify the linked list.

Examples:

Example 1:

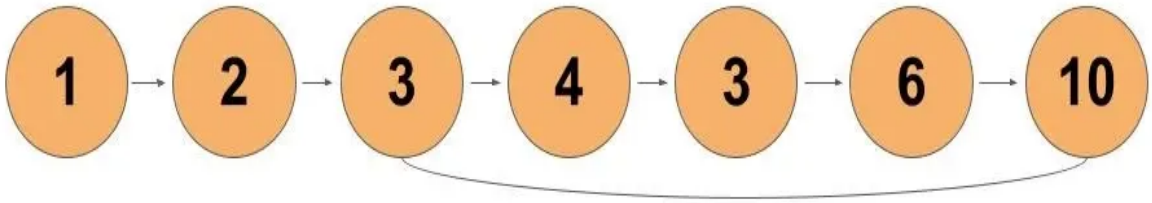
Input:

```
 head = [1,2,3,4,3,6,10]
```

Output:

```
 tail connects to node index 2
```

Explanation:



Example 2:

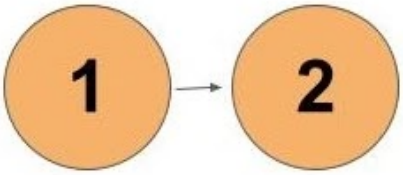
Input:

```
 head = [1,2]
```

Output:

```
 no cycle
```

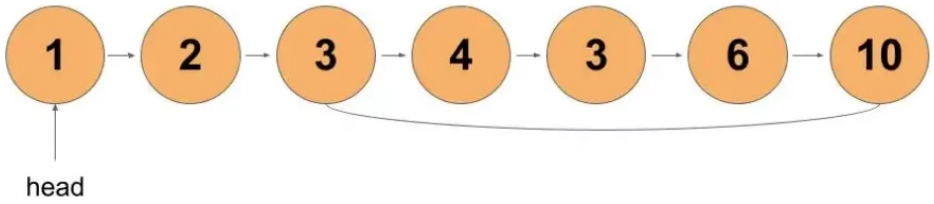
Explanation:

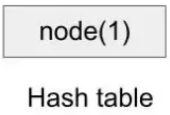


Solution:
Solution 1: Brute Force
Approach:
We can store nodes in a hash table so that, if a loop exists, the head will encounter the same node again. This node will be present in the table and hence, we can detect the loop. Steps are:-

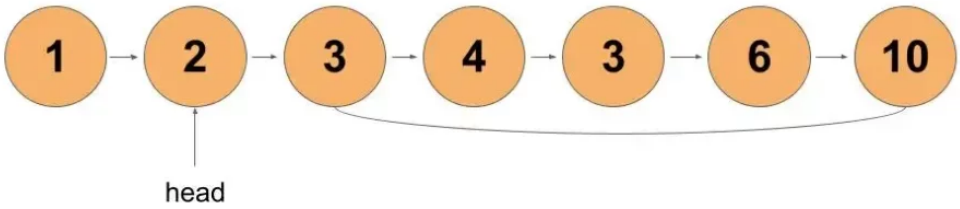
- Iterate the given list.
- For each node visited by head pointer, check if node is present in the hash table.
- If yes, loop detected
- If not, insert node in the hash table and move the head pointer ahead.
- If head reaches null, then the given list does not have a cycle in it.

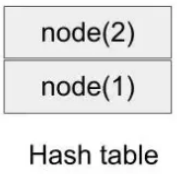
Dry Run:
We start iterating each node and storing nodes in the hash table if an element is not present.



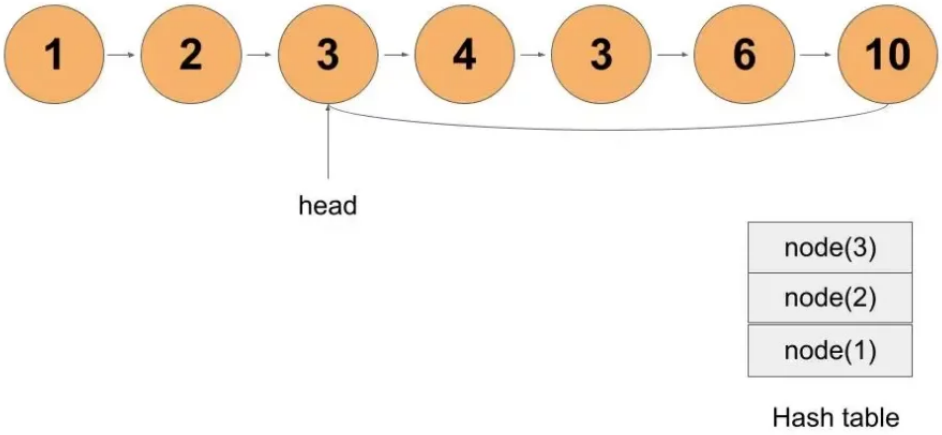


Node(1) is not present in the hash table. So, we insert a node in it and move head ahead.

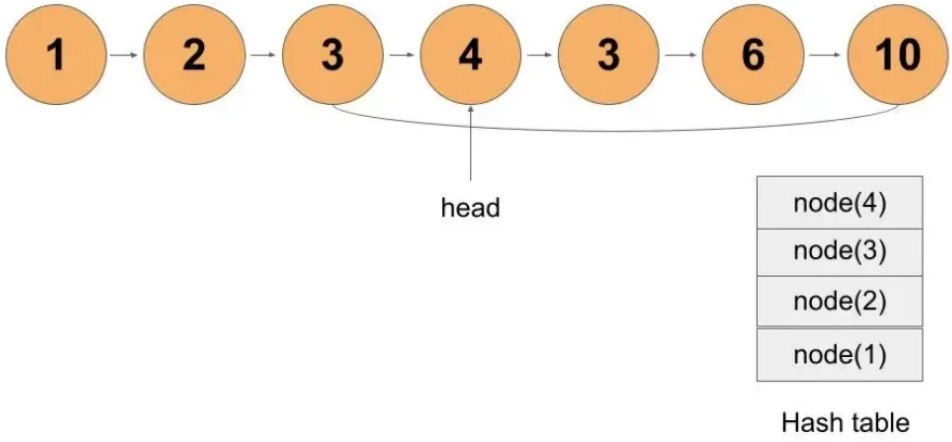




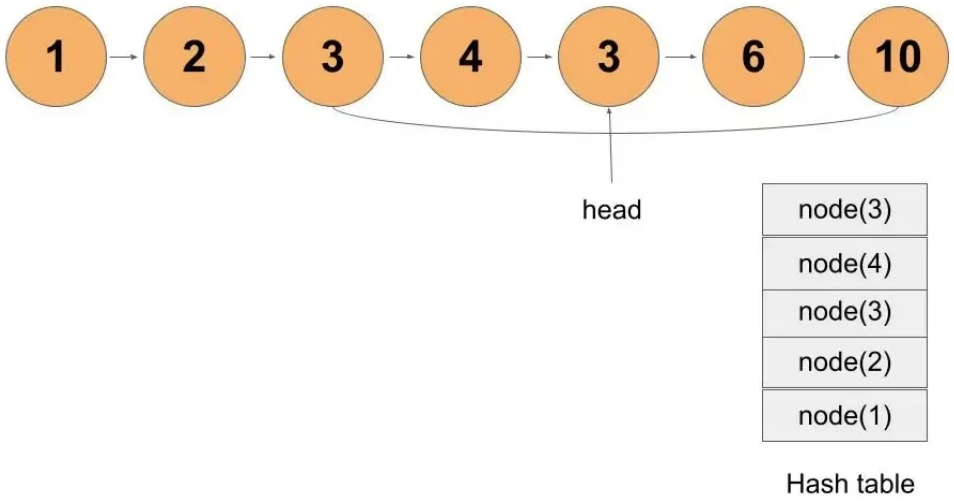
Node(2) is not present in the hash table. So, we insert a node in it and move head ahead.



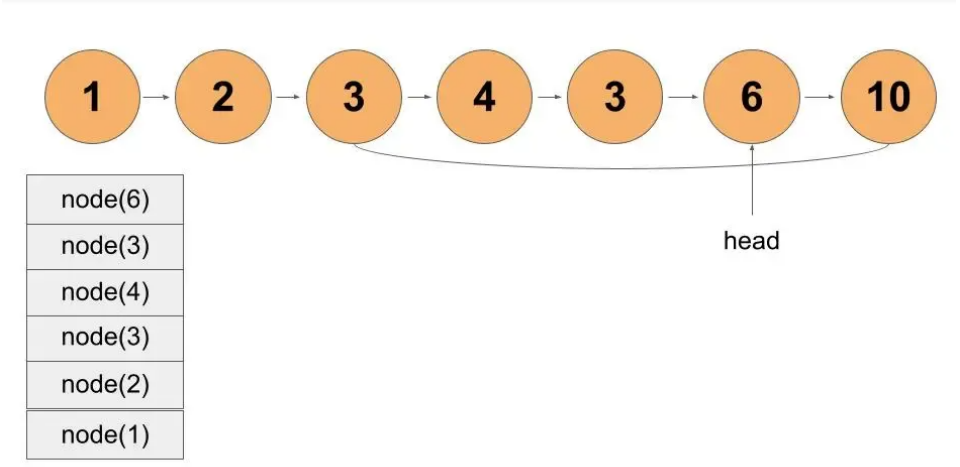
Node(3) is not present in the hash table. So, we insert a node in it and move head ahead.



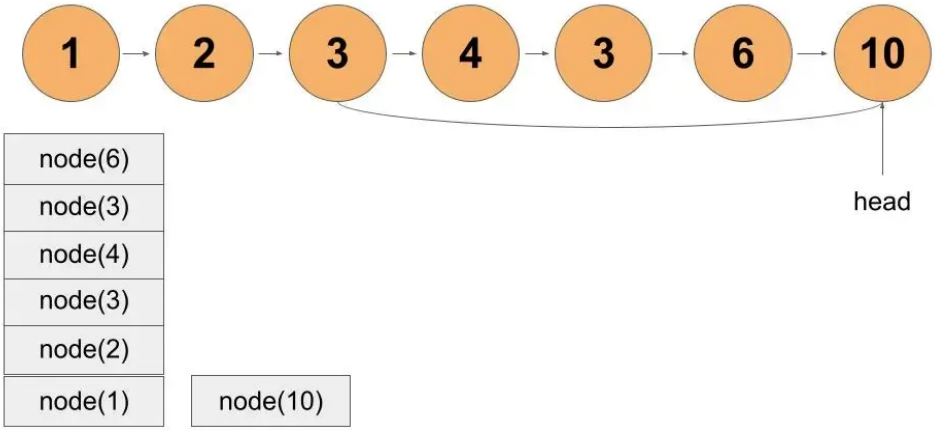
Node(4) is not present in the hash table. So, we insert a node in it and move head ahead.



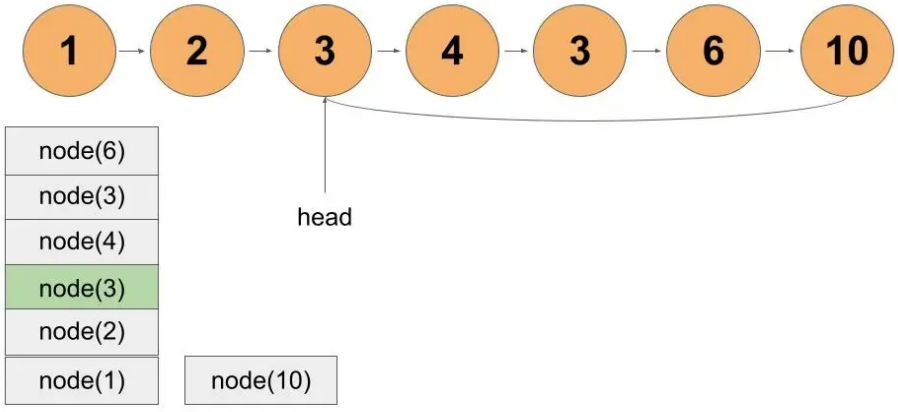
Node(3) is not present in the hash table. So, we insert a node in it and move head ahead. Though this node contains 3 as a value, it is a different node than the node at position 3.



Node(6) is not present in the hash table. So, we insert a node in it and move head ahead.



Node(10) is not present in the hash table. So, we insert a node in it and move head ahead.



We reached the same node which was present in the hash table. Thus, the starting node of the cycle is node(3).

Code:

```
//process as per mentioned in solution
node* detectCycle(node* head) {
    unordered_set<node*> st;
    while(head != NULL) {
        if(st.find(head) != st.end()) return head;
        st.insert(head);
        head = head->next;
    }
    return NULL;
}
```

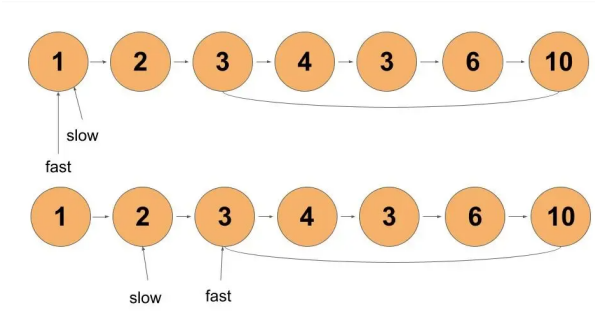
Output: Tail connects at pos 2
Time Complexity: O(N)
Reason: Iterating the entire list once.
Space Complexity: O(N)
Reason: We store all nodes in a hash table.

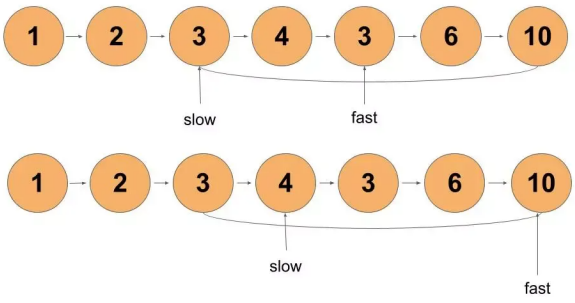
Solution 2: Slow and Fast Pointer Method
Approach:
The following steps are required:

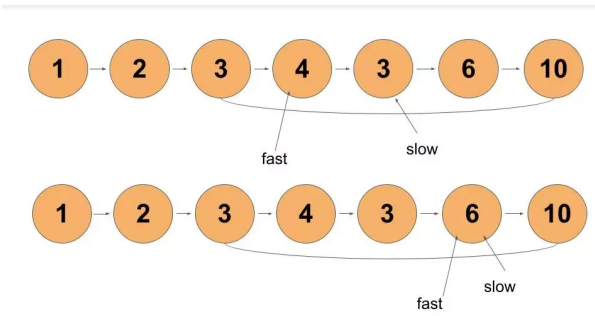
- Initially take two pointers, fast and slow. Fast pointer takes two steps ahead while slow pointer will take single step ahead for each iteration.
- We know that if a cycle exists, fast and slow pointers will collide.
- If cycle does not exists, fast pointer will move to NULL
- Else, when both slow and fast pointer collides, it detects a cycle exists.
- Take another pointer, say entry. Point to the very first of the linked list.
- Move the slow and the entry pointer ahead by single steps until they collide.
- Once they collide we get the starting node of the linked list.

But why use another pointer, entry?
Let's say a slow pointer covers the L2 distance from the starting node of the cycle until it collides with a fast pointer. L1 be the distance traveled by the entry pointer to the starting node of the cycle. So, in total, the slow pointer covers the L1+L2 distance. We know that a fast pointer covers some steps more than a slow pointer. Therefore, we can say that a fast pointer will surely cover the L1+L2 distance. Plus, a fast pointer will cover more steps which will accumulate to nC length where cC is the length of the cycle and n is the number of turns. Thus, the fast pointer covers the total length of L1+L2+nC.
We know that the slow pointer travels twice the fast pointer. So this makes equation to
2(L1+L2) = L1+L2+nC. This makes equation to
L1+L2 = nC. Moving L2 to the right side
L1 = nC-L2 and this shows why the entry pointer and the slow pointer would collide.

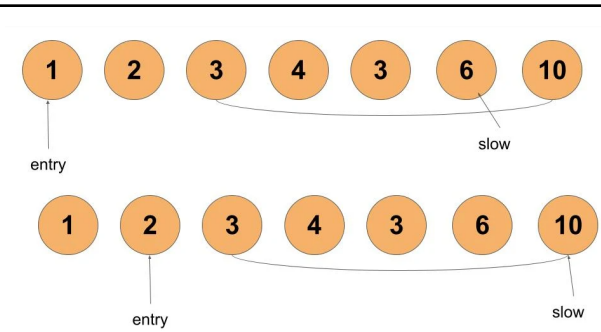
Dry Run:
We initialize fast and slow pointers to the head of the list. Fast moves two steps ahead and slowly takes a single step ahead.

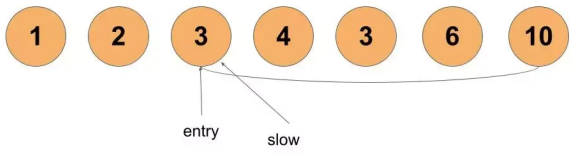






We can see that the fast and slow pointer collides which shows the cycle exists. The entry pointer is pointed to the head of the list. And move them forward until it collides with the slow pointer.





We see that both collide and hence, we get the starting node of the list.

Code:

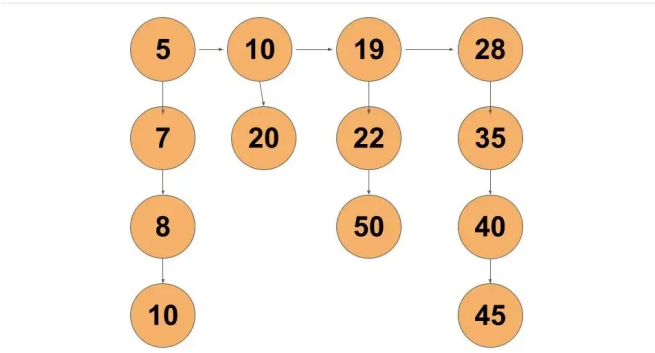
```
//process as per mentioned in solution
node* detectCycle(node* head) {
    if(head == NULL||head->next == NULL) return NULL;

    node* fast = head;
    node* slow = head;
    node* entry = head;

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

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

Output: Tail connects at pos 2
Time Complexity: O(N)
Reason: We can take overall iterations club it to O(N)
Space Complexity: O(1)
Reason: No extra data structure is used.

Flattening a Linked List

Flattening a Linked List
Problem Statement: Given a Linked List of size N, where every node represents a sub-linked-list and contains two pointers:
(i) a next pointer to the next node,
(ii) a bottom pointer to a linked list where this node is head.
Each of the sub-linked-list is in sorted order.
Flatten the Link List such that all the nodes appear in a single level while maintaining the sorted order.
Note: The flattened list will be printed using the bottom pointer instead of the next pointer.
Examples:
Example 1:
Input:

```
Number of head nodes = 4
Array holding length of each list with head and bottom = [4,2,3,4]
Elements of entire linked list = [5,7,8,30,10,20,19,22,50,28,35,40,45]
```



Output:

```
 Flattened list = [5,7,8,10,19,20,22,28,30,35,40,45,50]
```

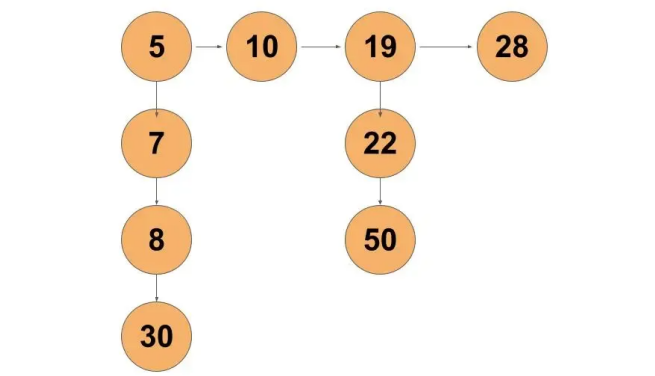
Explanation:

```
 Flattened list is the linked list consisting entire elements of the given list
in sorted order
```

Example 2:

Input:

```
Number of head nodes = 4
Array holding length of each list with head and bottom = [4,1,3,1]
Elements of entire linked list = [5,7,8,30,10,19,22,50,28]
```



Output:

```
 Flattened list = [5,7,8,10,19,22,28,30,50]
```

Explanation:

```
 Flattened list is the linked list consisting entire elements of the given list
in sorted order
```

Solution:

Approach:

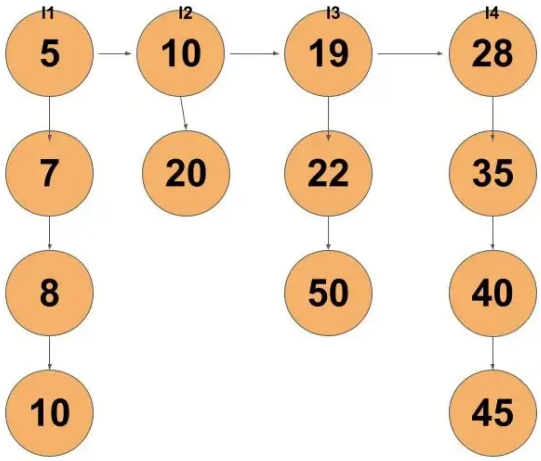
Since each list, followed by the bottom pointer, are in sorted order. Our main aim is to make a single list in sorted order of all nodes. So, we can think of a merge algorithm of merge sort.

The process to flatten the given linked list is as follows:-

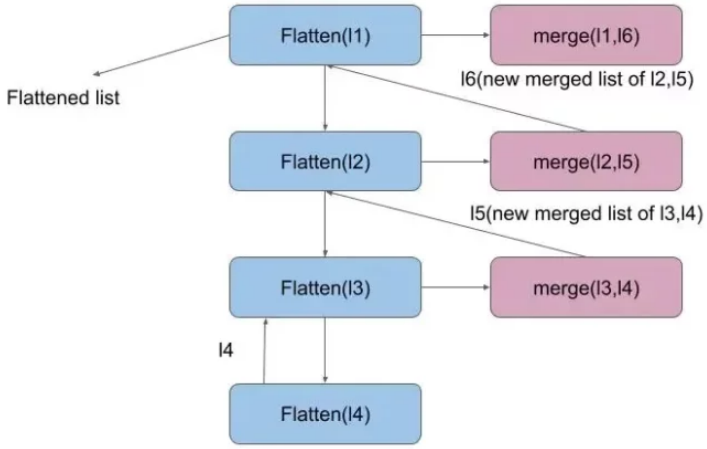
- We will recurse until the head pointer moves null. The main motive is to merge each list from the last.
- Merge each list chosen using the merge algorithm. The steps are
- Create a dummy node. Point two pointers, i.e, temp and res on dummy node. res is to keep track of dummy node and temp pointer is to move ahead as we build the flatten list.
- We iterate through the two chosen. Move head from any of the chosen list ahead whose current pointed node is smaller.
- Return the new flattened list found.

Dry Run:

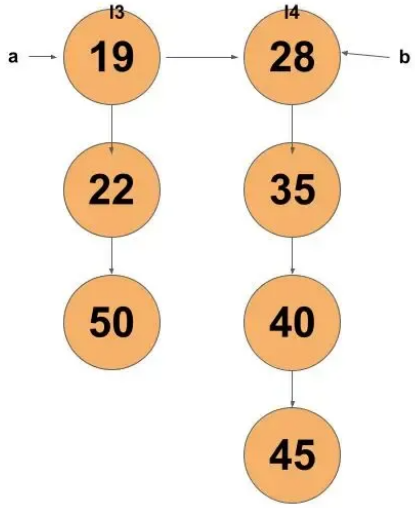
We will assign individual lists with bottom pointers names as l1, l2, l3, and l4 respectively for our convenience.

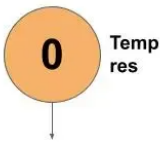


Let's see the recursion tree of the function flatten and merge function. It will trace down to allow the merge function to merge two sorted lists from the end.

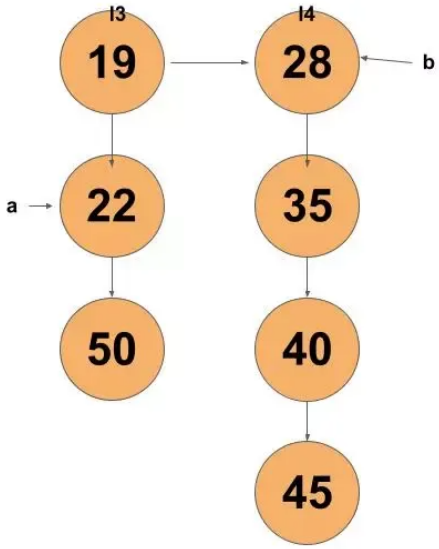


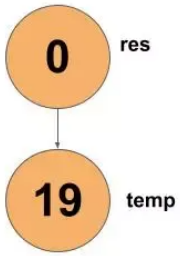
The merge function works like the merge algorithm of merge sort. Firstly, the algorithm will merge l3,l4 lists.



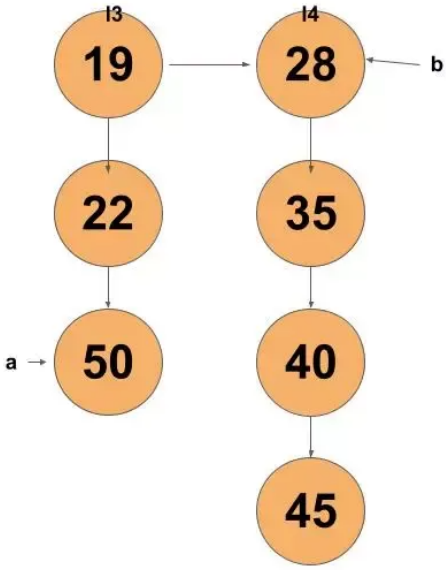


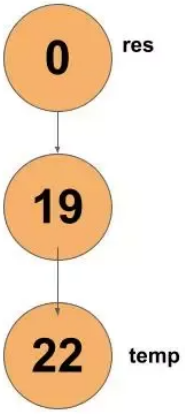
Node pointed by a is less than node pointed by b. So move node a and move temp ahead



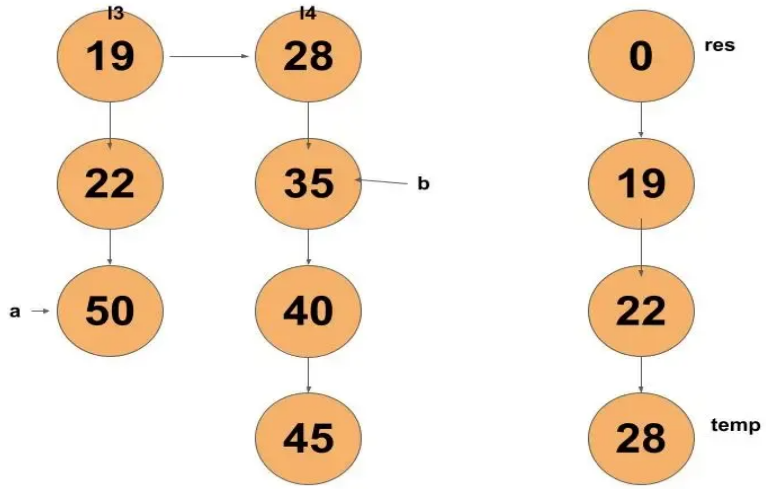


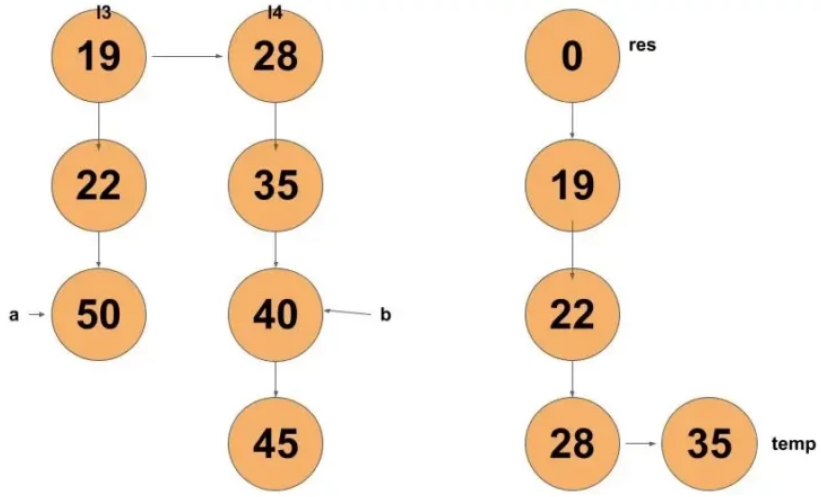
Node pointed by a is less than node pointed by b. So move node a and move temp ahead

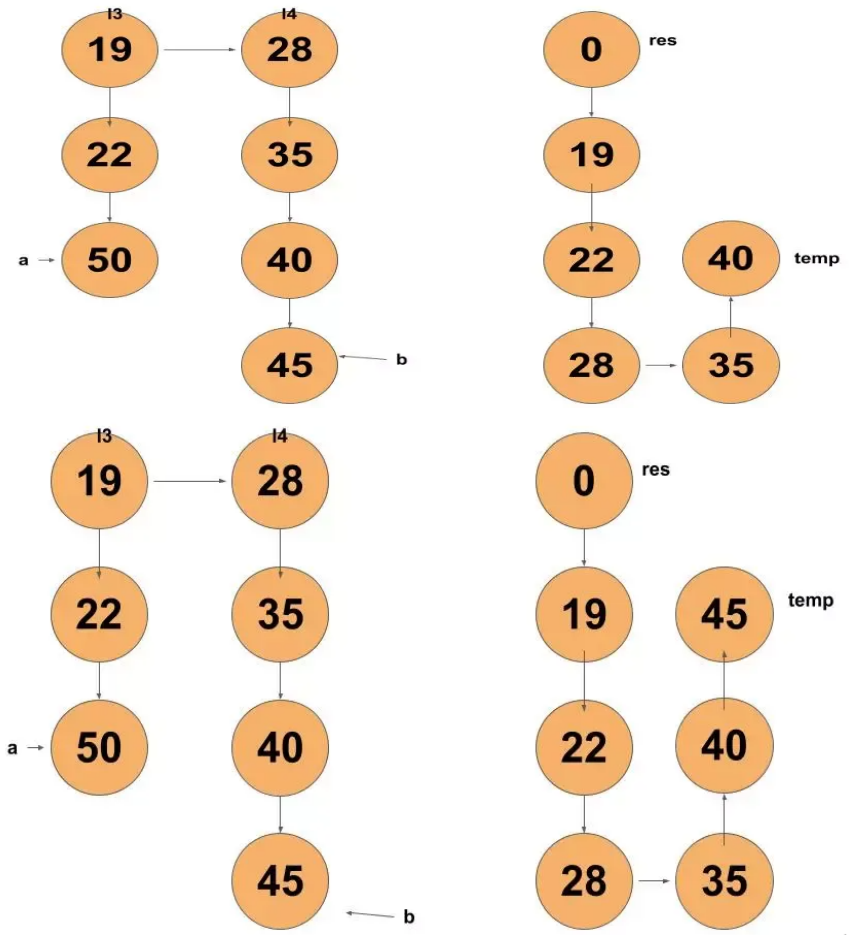




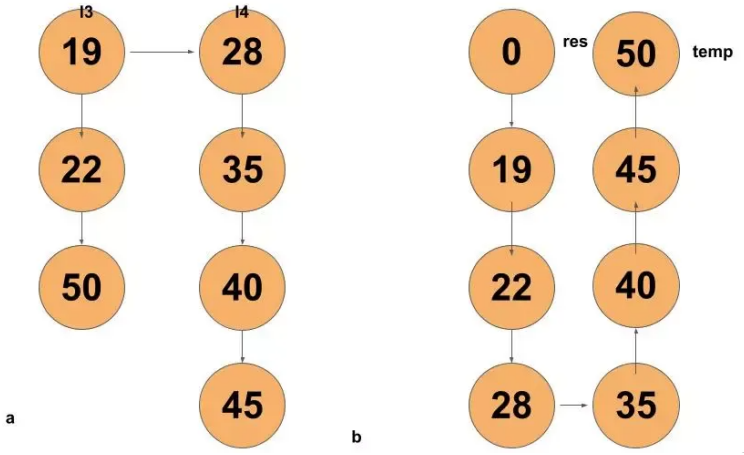
Node pointed by b is less than node pointed by a. So move node a and move temp ahead


l3
19
28
l4
22
35
b
a
50
40
45
0
res
19
22
28
temp


l3
19
28
l4
22
35
a
50
40
b
45
0
res
19
22
28
35
temp



Now, pointer b is null. So, we will merge the remaining nodes of pointer a until node a reaches null.



The same way other pairs of lists will be merged.

Code:

```
Node* mergeTwoLists(Node* a, Node* b) {

    Node *temp = new Node(0);
    Node *res = temp;

    while(a != NULL && b != NULL) {
        if(a->data < b->data) {
            temp->bottom = a;
            temp = temp->bottom;
            a = a->bottom;
        }
        else {
            temp->bottom = b;
            temp = temp->bottom;
            b = b->bottom;
        }
    }

    if(a) temp->bottom = a;
    else temp->bottom = b;

    return res -> bottom;

}
Node *flatten(Node *root)
{

        if (root == NULL || root->next == NULL)
            return root;

        // recur for list on right
        root->next = flatten(root->next);

        // now merge
        root = mergeTwoLists(root, root->next);

        // return the root
        // it will be in turn merged with its left
        return root;
}
```

Time Complexity: O(N), where N is the total number of nodes present
Reason: We are visiting all the nodes present in the given list.
Space Complexity: O(1)
Reason: We are not creating new nodes or using any other data structure.