**Question 1: Delete middle node of a Linked List**

Given a singly linked list, delete middle of the linked list.

For example, if given linked list is 1->2->3->4->5 then linked list should be modified to 1->2->4->5

If there are even nodes, then there would be two middle nodes, we need to delete the second middle element.

For example, if given linked list is 1->2->3->4->5->6 then it should be modified to 1->2->3->5->6.

Return the head of the linked list after removing the middle node.

If the input linked list has 1 node, then this node should be deleted and a null node should be returned.

Logic1 –

1. Find the length of the linked list.
2. If Length is 1 then delete the Head.
3. Then traverse till the middle element – 1 (length/2). Link this element to the next to next element, so that next element is deleted.

Logic2 –

1. Use 2 pointers slow and fast. Slow pointer moves 1 step at a time and fast pointer moves 2 steps at a time.
2. When the fast variable reaches the end of the linked list, the slow pointer will be pointing to the middle element.

Code –

/\*\*

 \* Definition for singly-linked list.

 \* class ListNode {

 \*     public int val;

 \*     public ListNode next;

 \*     ListNode(int x) { val = x; next = null; }

 \* }

 \*/

public class Solution {

    public ListNode solve(ListNode A) {

        if (A == null) {

            return A;

        }

        ListNode temp = A;

        int k = 0;

        while(temp != null) {

            temp = temp.next;

            k += 1;

        }

        // If len = 1

        if (k==1) {

            A = null;

            return A;

        }

        k/=2;

        temp = A;

        while(k > 1) {

            temp = temp.next;

            k-=1;

        }

        temp.next = temp.next.next;

        return A;

    }

}

**Question 2: K reverse linked list**

Given a singly linked list **A** and an integer **B**, reverse the nodes of the list **B** at a time and return the modified linked list.

**Problem Constraints**

1 <= |A| <= 103

B always divides A

Logic – Use Recursion

1. Reverse one chunk of length B then link the current head to the head of the next chunk.
2. Head of the next chunk can be found by recursively calling the function.
3. We can reverse the linked list by using 3 pointers. Below is the algorithm for reversing chunk of B elements.
4. Assign head to a variable “curr”.
5. Assign null to variable “prev”. prev will store the previous node of curr.
6. Store the node next to curr in a variable temp.
7. Link the curr node to prev.
8. Assign prev to curr and curr to temp. Here 1 link is reversed and we have moved forward to reverse next links.
9. Decrease k and repeat all steps from 3 till curr!=1 and k>0.

Code –

/\*\*

 \* Definition for singly-linked list.

 \* class ListNode {

 \*     public int val;

 \*     public ListNode next;

 \*     ListNode(int x) { val = x; next = null; }

 \* }

 \*/

public class Solution {

    public ListNode reverseList(ListNode A, int B) {

        if (A==null) {

            return null;

        }

        ListNode curr = A;

        ListNode prev = null;

        ListNode temp;

        int k = B;

        while(curr!=null && k>0) {

            temp = curr.next;  // Store the next variable in temp

            curr.next = prev;  // Reverse the first link

            prev = curr;       // Move Forward prev

            curr = temp;       // Move Forward curr

            k-=1;

        }

        A.next = reverseList(curr, B); // Link the current head to the head of next sorted chunk

        return prev; // return Head

    }

}

**Question 3: Middle element of linked list**

Given a linked list of integers, find and return the middle element of the linked list.

**NOTE:** If there are **N** nodes in the linked list and N is even then return the (N/2 + 1)th element.

**Problem Constraints**

1 <= length of the linked list <= 100000

1 <= Node value <= 109

Logic –

1. Find the length of the linked list.
2. Iterate till length/2 and find the middle element.

Code –

/\*\*

 \* Definition for singly-linked list.

 \* class ListNode {

 \*     public int val;

 \*     public ListNode next;

 \*     ListNode(int x) { val = x; next = null; }

 \* }

 \*/

public class Solution {

    public int solve(ListNode A) {

        ListNode temp = A;

        int k = 0;

        while(temp != null) {

            temp = temp.next;

            k+=1;

        }

        if (k==1) {

            return A.val;

        }

        k/=2;

        temp = A;

        while(k>1) {

            temp = temp.next;

            k-=1;

        }

        return temp.next.val;

    }

}

**Question 4: Reverse Link List II**

Reverse a linked list **A** from position **B** to **C**.

**NOTE:** Do it in-place and in one-pass.  
  
**Problem Constraints**

1 <= |A| <= 106

1 <= B <= C <= |A|

Logic –

1. Iterate till we find Bth element.
2. Store the Bth and the B-1th element in a variable for further use.
3. Start reversing from B to C elements (Same as reversing chunk of length C-B).
4. Link B-1th element to Head of the reversed chunk and Bth element to the next element in the linked list.

Code –

/\*\*

 \* Definition for singly-linked list.

 \* class ListNode {

 \*     public int val;

 \*     public ListNode next;

 \*     ListNode(int x) { val = x; next = null; }

 \* }

 \*/

public class Solution {

    public ListNode reverseBetween(ListNode A, int B, int C) {

        if (A==null) {return A;}

        if (A.next == null) {

            return A;

        }

        ListNode prev = null;

        ListNode curr = A;

        int k = B;

        while(curr !=null && k>1) {

            prev = curr;

            curr = curr.next;

            k-=1;

        }

        ListNode temp2 = prev;

        ListNode temp3 = curr;

        prev = curr;

        curr = curr.next;

        k = C-B;

        ListNode temp;

        while(curr!=null && k>0) {

            temp = curr.next;

            curr.next = prev;

            prev = curr;

            curr = temp;

            k-=1;

        }

        temp3.next = curr;

        if (temp2 != null) {

            temp2.next = prev;

        } else {

            A = prev;

        }

        return A;

    }

}

**Question 5: Reverse Linked List**

You are given a singly linked list having head node **A**. You have to reverse the linked list and return the head node of that reversed list.

**NOTE:** You have to do it **in-place** and in **one-pass**.  
  
**Problem Constraints**

1 <= Length of linked list <= 105

Value of each node is within the range of a 32-bit integer.

Logic – We can reverse the linked list by using 3 pointers.

1. Assign head to a variable “curr”.
2. Assign null to variable “prev”. prev will store the previous node of curr.
3. Store the node next to curr in a variable temp.
4. Link the curr node to prev.
5. Assign prev to curr and curr to temp. Here 1 link is reversed and we have moved forward to reverse next links.
6. Decrease k and repeat all steps from 3 till curr!=1.

Code –

/\*\*

 \* Definition for singly-linked list.

 \* class ListNode {

 \*     public int val;

 \*     public ListNode next;

 \*     ListNode(int x) { val = x; next = null; }

 \* }

 \*/

public class Solution {

    public ListNode reverseList(ListNode A) {

        if (A==null || A.next==null) {

            return A;

        }

        ListNode prev = null;

        ListNode curr = A;

        ListNode temp;

        while(curr != null) {

            temp = curr.next;

            curr.next = prev;

            prev = curr;

            curr = temp;

        }

        return prev;

    }

}

**Question 6: Design Linked List**

Design and implement a Linked List data structure.  
A node in a linked list should have the following attributes - an integer **value** and a **pointer** to the next node. It should support the following operations:

insert\_node(position, value) - To insert the input value at the given position in the linked list.

delete\_node(position) - Delete the value at the given position from the linked list.

print\_ll() - Print the entire linked list, such that each element is followed by a single space.

**Note:**

* If an input position does not satisfy the constraint,**no action** is required.
* Each print query has to be executed in a new line.

**Problem Constraints**

1 <= value <= 105  
1 <= position <= n where, n is the size of the linked-list.

Logic – We will maintain the head of the LinkedList.

1. For Insert operation - Firstly, we will traverse the list and keep two pointers, current and previous.  
   So if the position is 1, we will add the node in the beginning and update the head.  
   Otherwise, we will traverse the list up to the desired position and add the node by making the current node, the next node of the newly added node, and the next node of the previous node will be the newly added node.
2. For Print LinkedList Operation - We will print the data of all the nodes by traversing through the list and stop when our current pointer becomes null.
3. For Delete LinkedList Operation - We will traverse the list up to the desired position and keep track of two pointers, current and previous.  
   If the position is 1, we will make the new head of the list the next element of the last head. Otherwise, make the next element of the previous node the next element of the current node. At last, free the pointer of the current node.

Code –

public static class ListNode {

     public int val;

     public ListNode next;

     ListNode(int x) { val = x; next = null; }

  }

  public static ListNode Head = null;

  public static int length = 0;

    public static void insert\_node(int position, int value) {

        // @params position, integer

        // @params value, integer

        if (Head == null) {

            ListNode newNode = new ListNode(value);

            Head = newNode;

            length += 1;

            return;

        }

        if (position >= 1 && position <= length+1) {

            // Insert at head

            if(position == 1) {

                ListNode newNode = new ListNode(value);

                newNode.next = Head;

                Head = newNode;

                length += 1;

                return;

            }

            // Insert at tail

            if (position == length+1) {

                ListNode temp = Head;

                ListNode prev = null;

                while(temp != null) {

                    prev = temp;

                    temp = temp.next;

                }

                ListNode newNode = new ListNode(value);

                prev.next = newNode;

                length += 1;

                return;

            }

            // Other cases

            int k = position;

            ListNode temp = Head;

            ListNode prev = null;

            ListNode newNode = new ListNode(value);

            while(k>1) {

                prev = temp;

                temp = temp.next;

                k-=1;

            }

            prev.next = newNode;

            newNode.next = temp;

            length += 1;

            return;

        }

    }

    public static void delete\_node(int position) {

        // @params position, integer

        if (Head == null) {

            return;

        }

        if (position >= 1 && position <= length) {

            if (position == 1) {

                Head = Head.next;

                length -= 1;

                return;

            }

            if (position == length) {

                ListNode temp = Head;

                ListNode prev = null;

                while(temp.next != null) {

                    prev = temp;

                    temp = temp.next;

                }

                prev.next = null;

                length -= 1;

                return;

            }

            int k = position;

            ListNode temp = Head;

            ListNode prev = null;

            while(k>1) {

                prev = temp;

                temp = temp.next;

                k-=1;

            }

            prev.next = temp.next;

            length -= 1;

            return;

        }

    }

    public static void print\_ll() {

        // Output each element followed by a space

        ListNode temp = Head;

        int flag = 0;

        while(temp != null) {

            if (flag==0) {

                flag = 1;

                System.out.print(temp.val);

            }

            else {

                System.out.print(" " + temp.val);

            }

            temp = temp.next;

        }

        return;

    }

**Question 7: Remove Nth Node from List End**

Given a linked list **A**, remove the **B-th** node from the end of the list and return its head.

For example, Given linked list: 1->2->3->4->5, and B = 2. After removing the second node from the end, the linked list becomes 1->2->3->5.

**NOTE:** If **B** is greater than the size of the list, remove the first node of the list.

**NOTE:** Try doing it using constant additional space.  
  
**Problem Constraints**

1 <= |A| <= 106

Logic – Find out the length of the list in one go. Then you know the number of the node to be removed. Traverse to the node and remove it.

Code –

/\*\*

 \* Definition for singly-linked list.

 \* class ListNode {

 \*     public int val;

 \*     public ListNode next;

 \*     ListNode(int x) { val = x; next = null; }

 \* }

 \*/

public class Solution {

    public ListNode removeNthFromEnd(ListNode A, int B) {

        ListNode temp = A;

        ListNode prev = null;

        int k = 0;

        while(temp != null) {

            temp = temp.next;

            k+=1;

        }

        if (k<=B) {

            A = A.next;

            return A;

        }

        prev = null;

        temp = A;

        k-=B;

        while(k>0){

            prev = temp;

            temp = temp.next;

            k-=1;

        }

        prev.next = temp.next;

        return A;

    }

}

**Question 8:  Remove Duplicates from Sorted List**

Given a **sorted** linked list, delete all duplicates such that each element appears only once.  
  
**Problem Constraints**

0 <= length of linked list <= 106

Logic – Iterate the linked list with 2 pointers curr and prev. If value of curr node is equal to value of prev node, then delete the curr node else move forward.

Code –

/\*\*

 \* Definition for singly-linked list.

 \* class ListNode {

 \*     public int val;

 \*     public ListNode next;

 \*     ListNode(int x) { val = x; next = null; }

 \* }

 \*/

public class Solution {

    public ListNode deleteDuplicates(ListNode A) {

        if(A==null || A.next==null) {

            return A;

        }

        ListNode prev = A;

        ListNode temp = A.next;

        while(temp != null) {

            if (temp.val == prev.val) {

                prev.next = temp.next;

            } else {

                prev = temp;

            }

            temp = temp.next;

        }

        return A;

    }

}

**Question 9:  Remove Loop from Linked List**

You are given a linked list that contains a loop.  
You need to find the node, which creates a loop and break it by making the node point to NULL.  
**Problem Constraints**

1 <= number of nodes <= 1000

Try solving it using constant additional space.

Logic1 – One logic is to use Hash Maps and find the first element that gets repeated.

Logic2 – Using slow and fast pointers

1. Move the slow and fast pointers till they meet. Slow pointer moves 1 step at a time and fast pointer moves 2 step at a time.
2. Once we find the meeting point, initialize slow to Head of the linked list and keep the fast at the meeting point and run the pointer again, but this time move both the pointers 1 step at a time.
3. The pointers will meet at the start of the cycle. Maintain a variable (prev) that stores the previous element of the fast pointer at every iteration. When the pointers meet each other, point this variable to null(prev.next = null).

Code –

/\*\*

 \* Definition for singly-linked list.

 \* class ListNode {

 \*     public int val;

 \*     public ListNode next;

 \*     ListNode(int x) { val = x; next = null; }

 \* }

 \*/

public class Solution {

    public ListNode solve(ListNode A) {

        if (A==null || A.next==null) return A;

        ListNode slow = A;

        ListNode fast = A;

        int flag = 0;

        while(fast.next != null && fast.next.next != null) {

            slow = slow.next;

            fast = fast.next.next;

            if (fast==slow) {

                flag = 1;

                break;

            }

        }

        if (flag == 0) {

            return A;

        }

        slow = A;

        ListNode prev = fast;

        while(slow != fast) {

            prev = fast;

            slow = slow.next;

            fast = fast.next;

        }

        prev.next = null;

        return A;

    }

}

**Question 9:  Reorder List**

Given a singly linked list **A**

A: A0 → A1 → … → An-1 → An

reorder it to:

A0 → An → A1 → An-1 → A2 → An-2 → …

You must do this in-place without altering the nodes' values.

Logic –

1. Find the mid-point of the linked list using fast and slow pointers. Slow pointer moves one step at a time and fast pointer moves 2 steps at a time.
2. Reverse the right half of the linked list using the mid-point. Cut the current linked list at the mid-point.
3. Now, we have 2 linked list, one is the left half of the original linked list and second is the reverse of right linked list.
4. Merge these two linked list into 1, by taking one element from each.

Code –

/\*\*

 \* Definition for singly-linked list.

 \* class ListNode {

 \*     public int val;

 \*     public ListNode next;

 \*     ListNode(int x) { val = x; next = null; }

 \* }

 \*/

public class Solution {

    public ListNode reverseList(ListNode A) {

        if (A==null || A.next==null) return A;

        ListNode curr = A;

        ListNode prev = null;

        ListNode temp = null;

        while(curr != null) {

            temp = curr.next;

            curr.next = prev;

            prev = curr;

            curr = temp;

        }

        return prev;

    }

    public ListNode reorderList(ListNode A) {

        if (A==null || A.next==null || A.next.next==null) return A;

        ListNode slow = A;

        ListNode fast = A;

        // Find the mid point of linked list

        while(fast.next != null && fast.next.next != null) {

            slow = slow.next;

            fast = fast.next.next;

        }

        ListNode h2 = slow.next;

        slow.next = null;

        ListNode p1 = A;

        ListNode p2 = reverseList(h2);

        ListNode t2 = null;

        // Merge the reversed second half List and current list

        while(p2 != null) {

            t2 = p2.next;

            p2.next = p1.next;

            p1.next = p2;

            p1 = p1.next.next;

            p2 = t2;

        }

        return A;

    }

}

**Question 10:  Sort List**

Sort a linked list, **A** in **O(n log n)** time using constant space complexity.  
  
**Problem Constraints**

0 <= |A| = 105

Logic –

1. Divide the linked list into 2 halves.
2. Apply the sort function recursively on these 2 halves.
3. Now, merge the sorted linked lists returned by the functions, such that they form a sorted linked list.
4. We can merge the sorted linked lists by using 3 pointers.
5. Create a dummy node (p1) with any random value.
6. Store that node in a node “ans”.
7. Initialize 2 variables (currA and currB) to head of each linked list.
8. Start Linking the p1 node to the node with smallest value between currA and currB. Then move the node with the smallest value forward and also move the p1 node forward.
9. Continue 4th step till value of one of the node becomes null.
10. At the end of the loop, when one of the lists goes empty, include the remaining elements from the second list into the answer.

Code –

/\*\*

 \* Definition for singly-linked list.

 \* class ListNode {

 \*     public int val;

 \*     public ListNode next;

 \*     ListNode(int x) { val = x; next = null; }

 \* }

 \*/

public class Solution {

    public ListNode midNode(ListNode A) {

        // Returns mid node

        if (A==null || A.next==null) return A;

        ListNode slow = A;

        ListNode fast = A;

        while(fast.next != null && fast.next.next != null) {

            slow = slow.next;

            fast = fast.next.next;

        }

        return slow;

    }

    public ListNode mergeList(ListNode A, ListNode B) {

        // Merges 2 sorted lists to form a sorted linked list

        ListNode ans = new ListNode(-1);

        ListNode p1 = ans;

        ListNode currA = A;

        ListNode currB = B;

        while(currA != null && currB != null) {

            if (currA.val < currB.val) {

                p1.next = currA;

                currA = currA.next;

                p1 = p1.next;

            } else {

                p1.next = currB;

                currB = currB.next;

                p1 = p1.next;

            }

        }

        if (currA == null) {

            p1.next = currB;

        }

        if (currB == null) {

            p1.next = currA;

        }

        return ans.next;

    }

    public ListNode sortList(ListNode A) {

        if (A == null || A.next == null) return A;

        ListNode temp = midNode(A);

        ListNode h2 = temp.next;

        temp.next = null;

        // A and h2

        ListNode temp1 = sortList(A);

        ListNode temp2 = sortList(h2);

        return mergeList(temp1, temp2);

    }

}

**Question 11:   Palindrome List**

Given a singly linked list **A**, determine if it's a palindrome. Return **1** or **0,** denoting if it's a palindrome or not, respectively.  
  
**Problem Constraints**

1 <= |A| <= 105

Logic –

1. Split the linked list into 2 halves.
2. Reverse the latter half and compare it with the first half.

Code –

/\*\*

 \* Definition for singly-linked list.

 \* class ListNode {

 \*     public int val;

 \*     public ListNode next;

 \*     ListNode(int x) { val = x; next = null; }

 \* }

 \*/

public class Solution {

    public ListNode reverseList(ListNode A) {

        ListNode prev = null;

        ListNode curr = A;

        ListNode temp = null;

        while(curr != null) {

            temp = curr.next;

            curr.next = prev;

            prev = curr;

            curr = temp;

        }

        return prev;

    }

    public ListNode findMid(ListNode A) {

        ListNode slow = A;

        ListNode fast = A;

        while(fast.next != null && fast.next.next != null) {

            slow = slow.next;

            fast = fast.next.next;

        }

        return slow;

    }

    public int lPalin(ListNode Head) {

        ListNode mid = findMid(Head);

        ListNode p1 = mid.next;

        ListNode A = Head;

        p1 = reverseList(p1);

        while(p1 != null && A != null) {

            if(A.val != p1.val) {

                return 0;

            }

            p1 = p1.next;

            A = A.next;

        }

        return 1;

    }

}

**Question 12:   Longest Palindromic List**

Given a linked list of integers. Find and return the length of the longest palindrome list that exists in that linked list.

A palindrome list is a list that reads the same backward and forward.

Expected memory complexity : O(1)  
  
**Problem Constraints**

1 <= length of the linked list <= 2000

1 <= Node value <= 100

Logic –

1. Initialize 2 variable curr and prev to Head and null respectively.
2. Link the curr to the prev. Basically, start reversing the linked list while checking the length of longest pallidrome.
3. Considering curr as the mid of the pallidrome, count the length of the even and the odd palindrome. Compare it with the ans and store the maximum out of the 3 variables (ans, length of even palindrome and length of odd palindrome).
4. Move the variables curr and prev forward till curr becomes null.

Code –

/\*\*

 \* Definition for singly-linked list.

 \* class ListNode {

 \*     public int val;

 \*     public ListNode next;

 \*     ListNode(int x) { val = x; next = null; }

 \* }

 \*/

public class Solution {

    public int count(ListNode A, ListNode B) {

        int count = 0;

        ListNode temp1 = A;

        ListNode temp2 = B;

        while(temp1 != null && temp2 != null && temp1.val == temp2.val) {

            temp1 = temp1.next;

            temp2 = temp2.next;

            count += 1;

        }

        return count;

    }

    public int solve(ListNode A) {

        int count = 0;

        ListNode curr = A;

        ListNode prev = null;

        ListNode temp = null;

        while(curr != null) {

            temp = curr.next;

            curr.next = prev;

            // Count the pallidrome length

            int x = 2 \* count(curr, temp);       // Even pallidrome

            int y = 2 \* count(prev, temp) + 1;   // Odd pallidrome

            count = Math.max(count, x);

            count = Math.max(count, y);

            prev = curr;

            curr = temp;

        }

        return count;

    }

}

**Question 13:   LRU (Least Recently Used) Cache**

Design and implement a data structure for Least Recently Used (LRU) cache. It should support the following operations: get and set.

* get(key) - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.
* set(key, value) - Set or insert the value if the key is not already present. When the cache reaches its capacity, it should invalidate the least recently used item before inserting the new item.

The LRUCache will be initialized with an integer corresponding to its capacity. Capacity indicates the maximum number of unique keys it can hold at a time.

**Definition of "least recently used"** : An access to an item is defined as a get or a set operation of the item. "Least recently used" item is the one with the oldest access time.

**NOTE:** If you are using any global variables, make sure to clear them in the constructor.

Logic – Doubly Linked List and HashMaps

1. Create a class Node which stores key, value, address of previous node and next node.
2. Create 2 dummy nodes with dummy values, named head and tail. Connect head.next to tail and tail.prev to head. All the elements will come in between these 2 nodes.
3. Create a hashmap which will store key and the address of the node.
4. Store a variable size = 0, to store the size of the cache.
5. When we get an element in the cache, we get 2 cases:
6. The key is not present in the HashMap, then return -1.
7. If the key is present in the HashMap then remove that element and add it at the tail of the linked list.
8. When we have to set an element in the cache, we get 4 cases:
9. The key is not present in the Hashmap and size == capacity, then we have to remove the first inserted element (head.next) and then insert the new node at tail of the linked list.
10. The key is not present in Hashmap and size < capacity, then simply insert the new node at the tail of the linked list and increment size.
11. The key is present in the HashMap then we have to remove that element from the linked list and insert the new node at the tail of the linked list.

Code –

import java.util.Arrays;

public class Solution {

    // Doubly linked list

    public class Node {

        int val;   // Store the key

        int key;

        Node next;

        Node prev;

        public Node(int c, int d) {

            key = c;

            val = d;

            prev = null;

            next = null;

        }

    }

    int cap = 0;

    int size = 0;

    Node head = new Node(-1, -1);

    Node tail = new Node(-1, -1);

    // Node -> value

    HashMap<Integer, Node> map = new HashMap<>();

    public Solution(int capacity) {

        this.cap = capacity;

        this.head.next = this.tail;

        this.tail.prev = this.head;

    }

    public void insert(Node x) {

        x.next = tail;

        x.prev = tail.prev;

        tail.prev = x;

        x.prev.next = x;

    }

    public void remove(Node x) {

        x.prev.next = x.next;

        x.next.prev = x.prev;

    }

    public int get(int key) {

        // Check if present in HashMap

        if(map.containsKey(key)) {

            Node x = map.get(key);

            remove(x);

            insert(x);

            return x.val;

        }

        return -1;

    }

    public void set(int key, int value) {

        if(!map.containsKey(key)) {

            if (size == this.cap) {

                map.remove(head.next.key);

                remove(head.next);

                size-=1;

            }

            Node x = new Node(key, value);

            insert(x);

            map.put(key, x);

            size+=1;

        }

        else {

            Node y = map.get(key);

            remove(y);

            map.remove(key);

            Node x = new Node(key, value);

            insert(x);

            map.put(key, x);

        }

    }

}

**Question 14:    Partition List**

Given a linked list **A** and a value **B**, partition it such that all nodes less than **B** come before nodes greater than or equal to **B**.

You should preserve the original relative order of the nodes in each of the two partitions.  
  
**Problem Constraints**

1 <= |A| <= 106

1 <= A[i], B <= 109

Logic –

1. Maintain 2 pointers such that the first pointer is the one that maintains all nodes that are less than x.
2. The second pointer is the one that maintains the nodes that are greater than or equal to x.
3. As we traverse along with our list, when we are at a node that is greater than or equal to x, we remove this node from our list and move it to a list of nodes that are greater than x.
4. At the last step we link the tail of smaller elements to the larger ones.

Code –

/\*\*

 \* Definition for singly-linked list.

 \* class ListNode {

 \*     public int val;

 \*     public ListNode next;

 \*     ListNode(int x) { val = x; next = null; }

 \* }

 \*/

public class Solution {

    public ListNode partition(ListNode A, int B) {

        if (A==null || A.next==null) return A;

        ListNode large = new ListNode(-1);

        ListNode p1 = large;

        ListNode temp = A;

        ListNode prev = null;

        while(temp != null) {

            if (temp.val >= B) {

                ListNode newNode = new ListNode(temp.val);

                p1.next = newNode;

                p1 = p1.next;

                if (prev == null) {

                    A = A.next;

                    temp = A;

                }

                else {

                    temp = temp.next;

                    prev.next = temp;

                }

            }

            else {

                prev = temp;

                temp = temp.next;

            }

        }

        if (prev == null) return large.next;

        prev.next = large.next;

        return A;

    }

}

**Question 15:    Flatten a linked list**

Given a linked list where every node represents a linked list and contains two pointers of its type:

1. Pointer to next node in the main list (**right pointer**)
2. Pointer to a linked list where this node is head (**down pointer**). All linked lists are sorted.

You are asked to flatten the linked list into a single list. Use **down** pointer to link nodes of the flattened list. **The flattened linked list should also be sorted.**  
  
**Problem Constraints**

1 <= Total nodes in the list <= 100000

1 <= Value of node <= 109

Logic – Recursion

1. We can use the merge function from merge sort. So the answer will be the merged list of the root and flattened list of the root.right.

Code –

/\*

class ListNode {

    int val;

    ListNode right, down;

    ListNode(int x) {

        val = x;

        right = down = null;

    }

}

\*/

ListNode merge (ListNode a, ListNode b) {

    ListNode ans = new ListNode(-1);

    ListNode p1 = ans;

    ListNode h1 = a;

    ListNode h2 = b;

    while(h1 != null && h2 != null) {

        if(h1.val < h2.val) {

            p1.down = h1;

            p1 = p1.down;

            h1 = h1.down;

        }

        else {

            p1.down = h2;

            p1 = p1.down;

            h2 = h2.down;

        }

    }

    if (h1 != null) p1.down = h1;

    if (h2 != null) p1.down = h2;

    return ans.down;

}

ListNode flatten(ListNode root) {

    if (root == null || root.right == null) {

        return root;

    }

    return merge(root, flatten(root.right));

}

**Question 16:    Balanced Paranthesis**

Given an expression string **A**, examine whether the pairs and the orders of “{“,”}”, ”(“,”)”, ”[“,”]” are correct in **A**.

Refer to the examples for more clarity.

**Problem Constraints**

1 <= |A| <= 100

Logic –

Code –

class Solution:

    # @param A : string

    # @return an integer

    def solve(self, A):

        dict1 = {

            '}': '{',

            ')': '(',

            ']': '['

        }

        arr = []

        for i in A:

            if i not in dict1:

                arr.append(i)

            elif i in dict1:

                if len(arr) == 0:

                    return 0

                if arr[-1] == dict1[i]:

                    arr.pop()

                else:

                    return 0

        if len(arr) == 0:

            return 1

        return 0

Day 58 59 60 61 62

Clone a Linked List

Intersection of 2 linked list

Sort a stack using another stack

Implement 2 stacks using array