**Linked list questions**

1. Write a method that receives the head of a sorted linked list then remove duplicates from it (The solution should be in O(N) time (

Ex.

Input: [ 1 2 2 5 5 5 5 8 9 9 9]

Output: [1 2 5 8 9]

**class** LinkedList

{

    Node head;  // head of list

    /\* Linked list Node\*/

**class** Node

    {

**int** data;

        Node next;

        Node(**int** d) {data = d; next = **null**; }

    }

**void** removeDuplicates()

    {

        /\*Another reference to head\*/

        Node curr = head;

        /\* Traverse list till the last node \*/

**while** (curr != **null**) {

             Node temp = curr;

            /\*Compare current node with the next node and

            keep on deleting them until it matches the current

            node data \*/

**while**(temp!=**null** && temp.data.equals(curr.data)) {

                temp = temp.next;

            }

            /\*Set current node next to the next different

            element denoted by temp\*/

            curr.next = temp;

            curr = curr.next;

        }

    }

    /\* Utility functions \*/

    /\* Inserts a new Node at front of the list. \*/

**public** **void** push(**int** new\_data)

    {

        /\* 1 & 2: Allocate the Node &

                  Put in the data\*/

        Node new\_node = **new** Node(new\_data);

        /\* 3. Make next of new Node as head \*/

        new\_node.next = head;

        /\* 4. Move the head to point to new Node \*/

        head = new\_node;

    }

    /\* Function to print linked list \*/

**void** printList()

     {

         Node temp = head;

**while** (temp != **null**)

         {

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

            temp = temp.next;

         }

         System.out.println();

     }

     /\* Driver program to test above functions \*/

**public** **static** **void** main(String args[])

    {

        LinkedList llist = **new** LinkedList();

        llist.push(20);

        llist.push(13);

        llist.push(13);

        llist.push(11);

        llist.push(11);

        llist.push(11);

        System.out.println("List before removal of duplicates");

        llist.printList();

        llist.removeDuplicates();

        System.out.println("List after removal of elements");

        llist.printList();

    }

}

1. Write a method that receives the head of two linked lists then find the intersection between them.

Ex.

Input:

[ 1 5 8 7 4 9 ]

[ 2 5 8 9 6 3 4 ]

Output: [ 5 8 4 9 ]

**class** GFG

{

    // head nodes for pointing to 1st and 2nd linked lists

**static** Node a = **null**, b = **null**;

    // dummy node for storing intersection

**static** Node dummy = **null**;

    // tail node for keeping track of

  // last node so that it makes easy for insertion

**static** Node tail = **null**;

    // class - Node

**static** **class** Node {

**int** data;

        Node next;

        Node(**int** data) {

**this**.data = data;

            next = **null**;

        }

    }

    // function for printing the list

**void** printList(Node start) {

        Node p = start;

**while** (p != **null**) {

            System.out.print(p.data + " ");

            p = p.next;

        }

        System.out.println();

    }

    // inserting elements into list

**void** push(**int** data) {

        Node temp = **new** Node(data);

**if**(dummy == **null**) {

            dummy = temp;

            tail = temp;

        }

**else** {

            tail.next = temp;

            tail = temp;

        }

    }

    // function for finding intersection and adding it to dummy list

**void** sortedIntersect()

    {

        // pointers for iterating

        Node p = a,q = b;

**while**(p != **null**  &&  q != **null**)

        {

**if**(p.data == q.data)

            {

                // add to dummy list

                push(p.data);

                p = p.next;

                q = q.next;

            }

**else** **if**(p.data < q.data)

                p = p.next;

**else**

                q= q.next;

        }

    }

  // Driver code

**public** **static** **void** main(String args[])

    {

        GFG list = **new** GFG();

        // creating first linked list

        list.a = **new** Node(1);

        list.a.next = **new** Node(2);

        list.a.next.next = **new** Node(3);

        list.a.next.next.next = **new** Node(4);

        list.a.next.next.next.next = **new** Node(6);

        // creating second linked list

        list.b = **new** Node(2);

        list.b.next = **new** Node(4);

        list.b.next.next = **new** Node(6);

        list.b.next.next.next = **new** Node(8);

        // function call for intersection

        list.sortedIntersect();

        // print required intersection

        System.out.println("Linked list containing common items of a & b");

        list.printList(dummy);

    }

}

1. Given a linked list, write a method to rearrange its nodes such that alternate positions are filled with nodes starting from the beginning and end of the list. **in linear time** (one node from beginning followed by one node from the end)

Ex.

Input: 1 —> 2 —> 3 —> 4 —> 5 —> 6  
Output: 1 —> 6 —> 2 —> 5 —> 3 —> 4

**class** GfG

{

// Linked List Node

**static** **class** Node

{

**int** data;

    Node next;

}

// A utility function to create a new node

**static** Node newNode(**int** key)

{

    Node temp = **new** Node();

    temp.data = key;

    temp.next = **null**;

**return** temp;

}

// Rearranges given linked list

// such that all even positioned

// nodes are before odd positioned.

// Returns new head of linked List.

**static** Node rearrangeEvenOdd(Node head)

{

    // Corner case

**if** (head == **null**)

**return** **null**;

    // Initialize first nodes of even and

    // odd lists

    Node odd = head;

    Node even = head.next;

    // Remember the first node of even list so

    // that we can connect the even list at the

    // end of odd list.

    Node evenFirst = even;

**while** (1 == 1)

    {

        // If there are no more nodes,

        // then connect first node of even

        // list to the last node of odd list

**if** (odd == **null** || even == **null** ||

                        (even.next) == **null**)

        {

            odd.next = evenFirst;

**break**;

        }

        // Connecting odd nodes

        odd.next = even.next;

        odd = even.next;

        // If there are NO more even nodes

        // after current odd.

**if** (odd.next == **null**)

        {

            even.next = **null**;

            odd.next = evenFirst;

**break**;

        }

        // Connecting even nodes

        even.next = odd.next;

        even = odd.next;

    }

**return** head;

}

// A utility function to print a linked list

**static** **void** printlist(Node node)

{

**while** (node != **null**)

    {

        System.out.print(node.data + "->");

        node = node.next;

    }

    System.out.println("NULL") ;

}

// Driver code

**public** **static** **void** main(String[] args)

{

    Node head = newNode(1);

    head.next = newNode(2);

    head.next.next = newNode(3);

    head.next.next.next = newNode(4);

    head.next.next.next.next = newNode(5);

    System.out.println("Given Linked List");

    printlist(head);

    head = rearrangeEvenOdd(head);

    System.out.println("Modified Linked List");

    printlist(head);

}

}

1. Given the head of a linked list, we repeatedly delete consecutive sequences of nodes that sum to 0 until there are no such sequences.

After doing so, return the head of the final linked list. You may return any such answer.

(Note that in the examples below, all sequences are serializations of ListNode objects.)

Example 1:

Input: head = [1,2,-3,3,1]  
Output: [3,1]  
Note: The answer [1,2,1] would also be accepted.

1. Reverse a linked list from position m to n. Do it in one-pass (The solution should be in O(N))

Ex.

Input: Linked list = [ 1 5 8 4 7 9 6 3] , m=2 n = 5

Output: [ 1 5 9 7 4 8 6 3 ]

**Note: No additional data structures is allowed**

class Solution {

public:

ListNode \*reverseBetween(ListNode \*head, int m, int n) {

ListNode\* newHead = new ListNode(-1);

newHead->next = head;

ListNode\* prev = newHead;

for(auto i = 0 ; i < m-1 ; i++){

prev = prev->next;

}

ListNode\* const reversedPrev = prev;

//position m

prev = prev->next;

ListNode\* cur = prev->next;

for(auto i = m ; i < n ; i++){

prev->next = cur->next;

cur->next = reversedPrev->next;

reversedPrev->next = cur;

cur = prev->next;

}

return newHead->next;}}

**class** LinkedList {

**static** Node head;

**static** **class** Node {

**int** data;

        Node next;

        Node(**int** d)

        {

            data = d;

            next = **null**;

        }

    }

    /\* Function to reverse the linked list \*/

**static** Node reverse(Node node)

    {

        Node prev = **null**;

        Node current = node;

**while** (current != **null**) {

            Node next = current.next;

            current.next = prev;

            prev = current;

            current = next;

        }

        node = prev;

**return** node;

    }

    // function used to reverse a linked list from position m to n

**static**  Node reverseBetween(Node head, **int** m, **int** n)

{

**if** (m == n)

**return** head;

    // revs and revend is start and end respectively of the

    // portion of the linked list which need to be reversed.

    // revs\_prev is previous of starting position and

    // revend\_next is next of end of list to be reversed.

    Node revs = **null**, revs\_prev = **null**;

    Node revend = **null**, revend\_next = **null**;

  +--+----------------------------

    // Find values of above pointers.

**int** i = 1;

    Node curr = head;

**while** (curr!=**null** && i <= n) {

**if** (i < m)

            revs\_prev = curr;

**if** (i == m)

            revs = curr;

**if** (i == n) {

            revend = curr;

            revend\_next = curr.next;

        }

        curr = curr.next;

        i++;

    }

    revend.next = **null**;

    // Reverse linked list starting with revs.

    revend = reverse(revs);

    // If starting position was not head

**if** (revs\_prev!=**null**)

        revs\_prev.next = revend;

    // If starting position was head

**else**

        head = revend;

    revs.next = revend\_next;

**return** head;

}

    // prints content of double linked list

**void** printList(Node node)

    {

**while** (node != **null**) {

            System.out.print(node.data + " ");

            node = node.next;

        }

    }

    // Driver Code

**public** **static** **void** main(String[] args)

    {

        LinkedList list = **new** LinkedList();

        list.head = **new** Node(10);

        list.head.next = **new** Node(20);

        list.head.next.next = **new** Node(30);

        list.head.next.next.next = **new** Node(40);

        list.head.next.next.next.next = **new** Node(50);

          list.head.next.next.next.next.next = **new** Node(60);

        list.head.next.next.next.next.next.next = **new** Node(70);

        reverseBetween(head,3,6);

        list.printList(head);

    }

}