**Linked list questions**

1. Write a method that receives the head of a linked list then check if the linked list is sorted or not. The method returns true if the linked list is sorted and false if not. The solution should be in O(N) time.

Ex. [1 2 3 5 8] 🡪 sorted

[2 1 5 4 ] 🡪 not sorted

**static** **class** Node

{

**int** data;

    Node next;

};

// function to Check Linked List is

// sorted in descending order or not

**static** **boolean** isSortedDesc(Node head){

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

**return** **true**;

    // Traverse the list till last node and return

    // false if a node is smaller than or equal

    // its next.

**for** (Node t = head; t.next != **null**; t = t.next)

**if** (t.data <= t.next.data)

**return** **false**;

**return** **true**;

}

**static** Node newNode(**int** data)

{

    Node temp = **new** Node();

    temp.next = **null**;

    temp.data = data;

**return** temp;

}

// Driver Code

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

{

    Node head = newNode(7);

    head.next = newNode(5);

    head.next.next = newNode(4);

    head.next.next.next = newNode(3);

**if**(isSortedDesc(head))

        System.out.println("Yes");

**else**

        System.out.println("No");

}

}

1. [Delete every N node in a linked list after skipping M nodes](https://www.techiedelight.com/delete-every-n-nodes-linked-list-skipping-m-nodes/) The solution should be in O(N2)time.

**Note: Not allowed to use any additional data structrues**

Ex.

Input: [ 1 2 8 5 4 7 9 5 9 6 7 8 ], N =3 ,M=2

Output: [1 2 7 9 7 8]

Explanation: go through the linked list as follows: skip m (here 2)nodes then delete N (here 3) nodes.

|  |
| --- |
| // Java program to delete N nodes  // after M nodes of a linked list  **import** java.util.\*;    **class** GFG  {    // A linked list node  **static** **class** Node  {  **int** data;      Node next;  };    /\* Function to insert a node at the beginning \*/  **static** Node push( Node head\_ref, **int** new\_data)  {      /\* allocate node \*/      Node new\_node = **new** Node();        /\* put in the data \*/      new\_node.data = new\_data;        /\* link the old list off the new node \*/      new\_node.next = (head\_ref);        /\* move the head to point to the new node \*/      (head\_ref) = new\_node;    **return** head\_ref;  }    /\* Function to print linked list \*/  **static** **void** printList( Node head){      Node temp = head;  **while** (temp != **null**)      {          System.out.printf("%d ", temp.data);          temp = temp.next;      }      System.out.printf("\n");  }    // Function to skip M nodes and then  // delete N nodes of the linked list.  **static** **void** skipMdeleteN( Node head, **int** M, **int** N)  {      Node curr = head, t;  **int** count;        // The main loop that traverses      // through the whole list  **while** (curr!=**null**)      {          // Skip M nodes  **for** (count = 1; count < M && curr != **null**; count++)              curr = curr.next;            // If we reached end of list, then return  **if** (curr == **null**)  **return**;            // Start from next node and delete N nodes          t = curr.next;  **for** (count = 1; count <= N && t != **null**; count++)          {              Node temp = t;              t = t.next;          }            // Link the previous list with remaining nodes          curr.next = t;            // Set current pointer for next iteration          curr = t;      }  }    // Driver code  **public** **static** **void** main(String args[])  {      /\* Create following linked list      1.2.3.4.5.6.7.8.9.10 \*/      Node head = **null**;  **int** M=2, N=3;      head=push(head, 10);      head=push(head, 9);      head=push(head, 8);      head=push(head, 7);      head=push(head, 6);      head=push(head, 5);      head=push(head, 4);      head=push(head, 3);      head=push(head, 2);      head=push(head, 1);        System.out.printf("M = %d, N = %d \nGiven" +                          "Linked list is :\n", M, N);      printList(head);        skipMdeleteN(head, M, N);        System.out.printf("\nLinked list after deletion is :\n");      printList(head);  }  } |

1. Write a method that receives the head of a linked list and an element, then move all occurrences of the element to end of linked list in O(N)

Ex. [1 2 5 4 1 1 7 8 1 9 ] , e=1

Output:

[2 5 4 7 8 9 1 1 1 1 ]

**class** GFG {

    // A Linked list Node

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

**int** data;

        Node next;

    }

    // A utility function to create a new node.

**static** Node newNode(**int** x)

    {

        Node temp = **new** Node();

        temp.data = x;

        temp.next = **null**;

**return** temp;

    }

    // Utility function to print the elements

    // in Linked list

**static** **void** printList(Node head)

    {

        Node temp = head;

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

            System.out.printf("%d ", temp.data);

            temp = temp.next;

        }

        System.out.printf("\n");

    }

    // Moves all occurrences of given key to

    // end of linked list.

**static** **void** moveToEnd(Node head, **int** key)

    {

        // Keeps track of locations where key

        // is present.

        Node pKey = head;

        // Traverse list

        Node pCrawl = head;

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

            // If current pointer is not same as pointer

            // to a key location, then we must have found

            // a key in linked list. We swap data of pCrawl

            // and pKey and move pKey to next position.

**if** (pCrawl != pKey && pCrawl.data != key) {

                pKey.data = pCrawl.data;

                pCrawl.data = key;

                pKey = pKey.next;

            }

            // Find next position where key is present

**if** (pKey.data != key)

                pKey = pKey.next;

            // Moving to next Node

            pCrawl = pCrawl.next;

        }

    }

    // Driver code

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

    {

        Node head = newNode(10);

        head.next = newNode(20);

        head.next.next = newNode(10);

        head.next.next.next = newNode(30);

        head.next.next.next.next = newNode(40);

        head.next.next.next.next.next = newNode(10);

        head.next.next.next.next.next.next = newNode(60);

        System.out.printf("Before moveToEnd(), the Linked list is\n");

        printList(head);

**int** key = 10;

        moveToEnd(head, key);

        System.out.printf("\nAfter moveToEnd(), the Linked list is\n");

        printList(head);

    }

}

1. Given a linked list, reverse the nodes of a linked list k at a time and return its 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. ( O(N) solution)

Ex.

Input:

[ 1 5 8 4 7 5 2 3 6 9 5 8 ], k= 3

The method should reverse every 3 nodes, the result should be

[8 5 1 5 7 4 6 3 2 8 5 9]

Ex2:

Input:

[5 4 8 7 2 8 9 6] , k=5

Output:

[2 7 8 4 5 8 9 6]

**class** LinkedList {

    Node head; // head of list

    /\* Linked list Node\*/

**class** Node {

**int** data;

        Node next;

        Node(**int** d)

        {

            data = d;

            next = **null**;

        }

    }

    Node reverse(Node head, **int** k)

    {

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

**return** **null**;

        Node current = head;

        Node next = **null**;

        Node prev = **null**;

**int** count = 0;

        /\* Reverse first k nodes of linked list \*/

**while** (count < k && current != **null**) {

            next = current.next;

            current.next = prev;

            prev = current;

            current = next;

            count++;

        }

        /\* next is now a pointer to (k+1)th node

           Recursively call for the list starting from

           current. And make rest of the list as next of

           first node \*/

**if** (next != **null**)

            head.next = reverse(next, k);

        // prev is now head of input list

**return** prev;

    }

    /\* 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();

        /\* Constructed Linked List is 1->2->3->4->5->6->

           7->8->8->9->null \*/

        llist.push(9);

        llist.push(8);

        llist.push(7);

        llist.push(6);

        llist.push(5);

        llist.push(4);

        llist.push(3);

        llist.push(2);

        llist.push(1);

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

        llist.printList();

        llist.head = llist.reverse(llist.head, 3);

        System.out.println("Reversed list");

        llist.printList();

    }

}

1. Given a linked list representation of two positive numbers, calculate and store their sum in a new list.

For example,

**Input:**  
   
X: 5 —> 7 —> 3 —> 4 —> null  
Y: 9 —> 4 —> 6 —> null  
   
**Output:** 6 —> 6 —> 8 —> 0 —> null  
   
(as 5734 + 946 = 6680)

**public** **class** linkedlistATN

{

**class** node

    {

**int** val;

        node next;

**public** node(**int** val)

        {

**this**.val = val;

        }

    }

    // Function to print linked list

**void** printlist(node head)

    {

**while** (head != **null**)

        {

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

            head = head.next;

        }

    }

    node head1, head2, result;

**int** carry;

    /\* A utility function to push a value to linked list \*/

**void** push(**int** val, **int** list)

    {

        node newnode = **new** node(val);

**if** (list == 1)

        {

            newnode.next = head1;

            head1 = newnode;

        }

**else** **if** (list == 2)

        {

            newnode.next = head2;

            head2 = newnode;

        }

**else**

        {

            newnode.next = result;

            result = newnode;

        }

    }

    // Adds two linked lists of same size represented by

    // head1 and head2 and returns head of the resultant

    // linked list. Carry is propagated while returning

    // from the recursion

**void** addsamesize(node n, node m)

    {

        // Since the function assumes linked lists are of

        // same size, check any of the two head pointers

**if** (n == **null**)

**return**;

        // Recursively add remaining nodes and get the carry

        addsamesize(n.next, m.next);

        // add digits of current nodes and propagated carry

**int** sum = n.val + m.val + carry;

        carry = sum / 10;

        sum = sum % 10;

        // Push this to result list

        push(sum, 3);

    }

    node cur;

    // This function is called after the smaller list is

    // added to the bigger lists's sublist of same size.

    // Once the right sublist is added, the carry must be

    // added to the left side of larger list to get the

    // final result.

**void** propogatecarry(node head1)

    {

        // If diff. number of nodes are not traversed, add carry

**if** (head1 != cur)

        {

            propogatecarry(head1.next);

**int** sum = carry + head1.val;

            carry = sum / 10;

            sum %= 10;

            // add this node to the front of the result

            push(sum, 3);

        }

    }

**int** getsize(node head)

    {

**int** count = 0;

**while** (head != **null**)

        {

            count++;

            head = head.next;

        }

**return** count;

    }

    // The main function that adds two linked lists

    // represented by head1 and head2. The sum of two

    // lists is stored in a list referred by result

**void** addlists()

    {

        // first list is empty

**if** (head1 == **null**)

        {

            result = head2;

**return**;

        }

        // first list is empty

**if** (head2 == **null**)

        {

            result = head1;

**return**;

        }

**int** size1 = getsize(head1);

**int** size2 = getsize(head2);

        // Add same size lists

**if** (size1 == size2)

        {

            addsamesize(head1, head2);

        }

**else**

        {

            // First list should always be larger than second list.

            // If not, swap pointers

**if** (size1 < size2)

            {

                node temp = head1;

                head1 = head2;

                head2 = temp;

            }

**int** diff = Math.abs(size1 - size2);

            // move diff. number of nodes in first list

            node temp = head1;

**while** (diff-- >= 0)

            {

                cur = temp;

                temp = temp.next;

            }

            // get addition of same size lists

            addsamesize(cur, head2);

            // get addition of remaining first list and carry

            propogatecarry(head1);

        }

            // if some carry is still there, add a new node to

            // the front of the result list. e.g. 999 and 87

**if** (carry > 0)

                push(carry, 3);

    }

    // Driver program to test above functions

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

    {

        linkedlistATN list = **new** linkedlistATN();

        list.head1 = **null**;

        list.head2 = **null**;

        list.result = **null**;

        list.carry = 0;

**int** arr1[] = { 9, 9, 9 };

**int** arr2[] = { 1, 8 };

        // Create first list as 9->9->9

**for** (**int** i = arr1.length - 1; i >= 0; --i)

            list.push(arr1[i], 1);

        // Create second list as 1->8

**for** (**int** i = arr2.length - 1; i >= 0; --i)

            list.push(arr2[i], 2);

        list.addlists();

        list.printlist(list.result);

    }

}