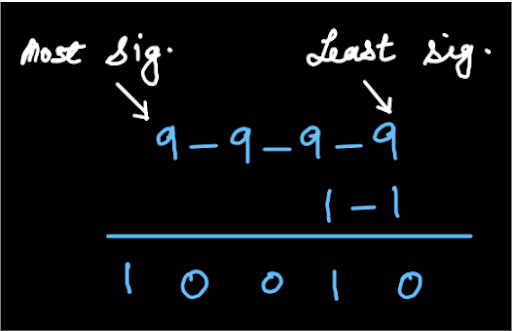
**1. Problem Discussion :**

You are required to write a function which is passed two linked lists and represents two numbers - the first element is the most significant digit and the last element is the least significant digit. The function is expected to add the two linked lists and return a new linked list.

****

DON'TS

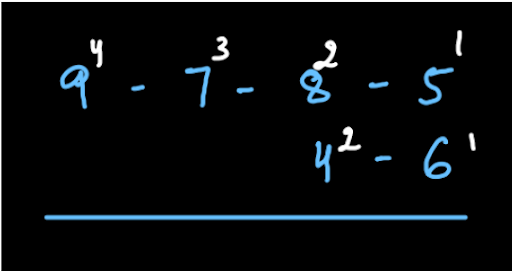
Don't reverse the linked lists in order to access them from least significant digit to most significant. Don't use arrays or explicit extra memory. Don't convert linked lists to numbers, add them up and convert the result back to a linked list.

CONSTRAINTS

Time complexity = O(n) Space complexity = Recursion space, O(n) This question is to be solved using the same trick we used in the previous questions, "Fold a linked list", "Is linked List a palindrome?" and "Display Reverse- Recursive".

**2. Approach :**

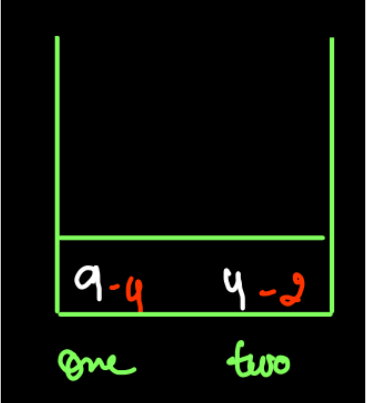
To solve this problem we will use a recursive approach. The idea here is that we need to add the list but as we know that addition is only possible when traversing from the end like adding firstly the unit place elements and then carry forwarding to the next place and so on. This generalized process here we will try to implement using recursion as we are not allowed to reverse the list. Let's see the solution through an example. We take the linked lists as 9-8-7-5 and 4-6. On top of each digit the place value of that digit is written in white.

****

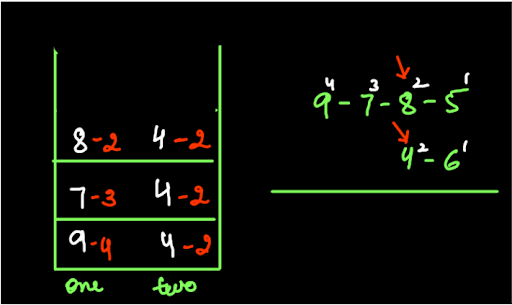
We use stack to understand this problem. It is divided into 2 parts: Going up the stack and Going down the stack.

UP THE STACK(PUSH)

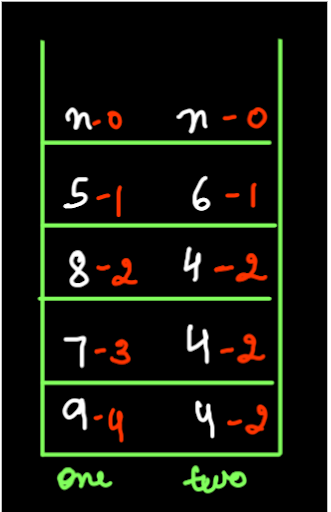
We make a stack and add the most significant digits of both the linked lists along with their place values (digits are denoted in white and place value in red in the figure

****

Since the place values of "one" and "two" are different they can't be added together. Hence we move towards the lesser significant digits for "one" till the place value of "one" and "two" is the same. We reach the position shown in figure where the place values are the same.

****

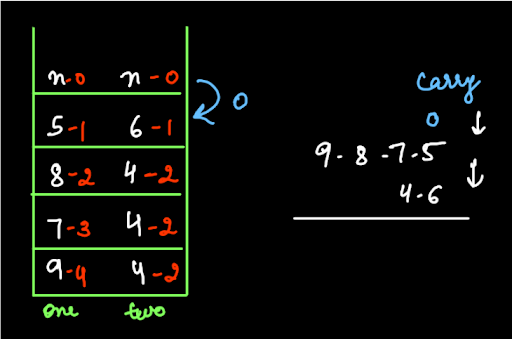
But we don't start the addition now because we have to start from the least significant digit and move towards the most significant digit. We move both the pointers forward and push a new level in the stack until they reach null

****

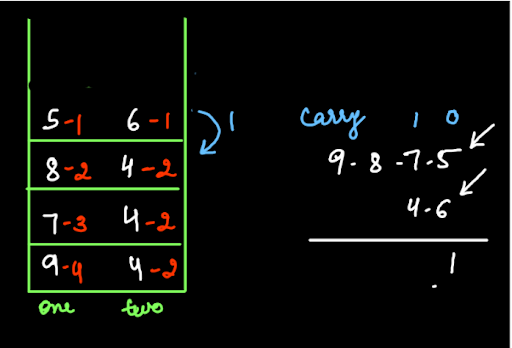
We reach the end of our pushing process.

DOWN THE STACK (POP)

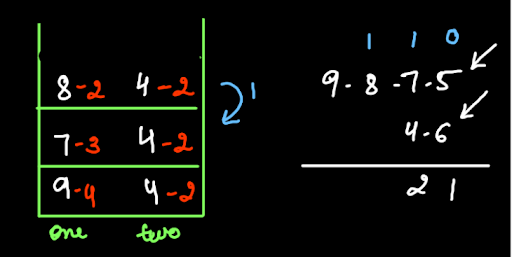
Now when we have reached the null position, we are going to return 0 as the carry value. This is our Base Condition.

****

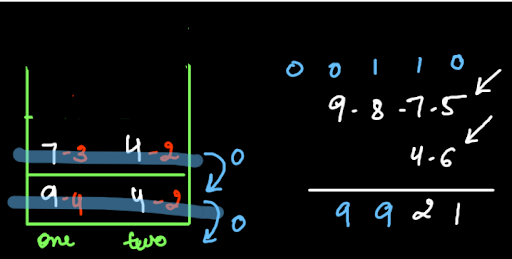
Now we take the sum of carry and the two digits. Here, the new carry is the ten's digit of the sum. This new carry is returned. Also, the one's digit of the sum is added first to the resultant linked list. In our example, 0+5+6=11. Hence ten's digit 1, the new carry is returned and one's digit, 1 is added to the resultant as shown in figure

****

As 1 is returned to the digits at place value 2, the process is repeated. Our sum =1+7+4=12. Now 1 is the new carry and 2 is added first to the resultant linked list.

****

As all the digits of the second linked list have been used up, therefore we just include carry and the next digit of the first linked list. So, 1+8=9. Hence, new carry is 0 and 9 is added first to the resultant. Again, for the greatest significant digit, 0+9=9 is the sum. Hence 0 is returned as carry and 9 is added first. \*Had the carry been more than 0 we would have added that value of carry in the resultant linked list too. Our final answer hence looks like as shown in figure

****

Hence our answer is 9921. Try to write the code for the above explanation yourself. If you face any problems, feel free to refer to the code given below.

**3. CODE:**

ConsoleJava

import java.io.\*;

import java.util.\*;

public class Main {

public static class Node {

int data;

Node next;

}

public static class LinkedList {

Node head;

Node tail;

int size;

void addLast(int val) {

Node temp = new Node();

temp.data = val;

temp.next = null;

if (size == 0) {

head = tail = temp;

} else {

tail.next = temp;

tail = temp;

}

size++;

}

public int size() {

return size;

}

public void display() {

for (Node temp = head; temp != null; temp = temp.next) {

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

}

System.out.println();

}

public void removeFirst() {

if (size == 0) {

System.out.println("List is empty");

} else if (size == 1) {

head = tail = null;

size = 0;

} else {

head = head.next;

size--;

}

}

public int getFirst() {

if (size == 0) {

System.out.println("List is empty");

return -1;

} else {

return head.data;

}

}

public int getLast() {

if (size == 0) {

System.out.println("List is empty");

return -1;

} else {

return tail.data;

}

}

public int getAt(int idx) {

if (size == 0) {

System.out.println("List is empty");

return -1;

} else if (idx < 0 || idx >= size) {

System.out.println("Invalid arguments");

return -1;

} else {

Node temp = head;

for (int i = 0; i < idx; i++) {

temp = temp.next;

}

return temp.data;

}

}

public void addFirst(int val) {

Node temp = new Node();

temp.data = val;

temp.next = head;

head = temp;

if (size == 0) {

tail = temp;

}

size++;

}

public void addAt(int idx, int val) {

if (idx < 0 || idx > size) {

System.out.println("Invalid arguments");

} else if (idx == 0) {

addFirst(val);

} else if (idx == size) {

addLast(val);

} else {

Node node = new Node();

node.data = val;

Node temp = head;

for (int i = 0; i < idx - 1; i++) {

temp = temp.next;

}

node.next = temp.next;

temp.next = node;

size++;

}

}

public void removeLast() {

if (size == 0) {

System.out.println("List is empty");

} else if (size == 1) {

head = tail = null;

size = 0;

} else {

Node temp = head;

for (int i = 0; i < size - 2; i++) {

temp = temp.next;

}

tail = temp;

tail.next = null;

size--;

}

}

public void removeAt(int idx) {

if (idx < 0 || idx >= size) {

System.out.println("Invalid arguments");

} else if (idx == 0) {

removeFirst();

} else if (idx == size - 1) {

removeLast();

} else {

Node temp = head;

for (int i = 0; i < idx - 1; i++) {

temp = temp.next;

}

temp.next = temp.next.next;

size--;

}

}

private Node getNodeAt(int idx) {

Node temp = head;

for (int i = 0; i < idx; i++) {

temp = temp.next;

}

return temp;

}

public void reverseDI() {

int li = 0;

int ri = size - 1;

while (li < ri) {

Node left = getNodeAt(li);

Node right = getNodeAt(ri);

int temp = left.data;

left.data = right.data;

right.data = temp;

li++;

ri--;

}

}

public void reversePI() {

if (size <= 1) {

return;

}

Node prev = null;

Node curr = head;

while (curr != null) {

Node next = curr.next;

curr.next = prev;

prev = curr;

curr = next;

}

Node temp = head;

head = tail;

tail = temp;

}

public int kthFromLast(int k) {

Node slow = head;

Node fast = head;

for (int i = 0; i < k; i++) {

fast = fast.next;

}

while (fast != tail) {

slow = slow.next;

fast = fast.next;

}

return slow.data;

}

public int mid() {

Node f = head;

Node s = head;

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

f = f.next.next;

s = s.next;

}

return s.data;

}

public static LinkedList mergeTwoSortedLists(LinkedList l1, LinkedList l2) {

LinkedList ml = new LinkedList();

Node one = l1.head;

Node two = l2.head;

while (one != null && two != null) {

if (one.data < two.data) {

ml.addLast(one.data);

one = one.next;

} else {

ml.addLast(two.data);

two = two.next;

}

}

while (one != null) {

ml.addLast(one.data);

one = one.next;

}

while (two != null) {

ml.addLast(two.data);

two = two.next;

}

return ml;

}

public static Node midNode(Node head, Node tail) {

Node f = head;

Node s = head;

while (f != tail && f.next != tail) {

f = f.next.next;

s = s.next;

}

return s;

}

public static LinkedList mergeSort(Node head, Node tail) {

if (head == tail) {

LinkedList br = new LinkedList();

br.addLast(head.data);

return br;

}

Node mid = midNode(head, tail);

LinkedList fsh = mergeSort(head, mid);

LinkedList ssh = mergeSort(mid.next, tail);

LinkedList sl = mergeTwoSortedLists(fsh, ssh);

return sl;

}

public void removeDuplicates() {

LinkedList res = new LinkedList();

while (this.size() > 0) {

int val = this.getFirst();

this.removeFirst();

if (res.size() == 0 || val != res.tail.data) {

res.addLast(val);

}

}

this.head = res.head;

this.tail = res.tail;

this.size = res.size;

}

public void oddEven() {

LinkedList odd = new LinkedList();

LinkedList even = new LinkedList();

while (this.size > 0) {

int val = this.getFirst();

this.removeFirst();

if (val % 2 == 0) {

even.addLast(val);

} else {

odd.addLast(val);

}

}

if (odd.size > 0 && even.size > 0) {

odd.tail.next = even.head;

this.head = odd.head;

this.tail = even.tail;

this.size = odd.size + even.size;

} else if (odd.size > 0) {

this.head = odd.head;

this.tail = odd.tail;

this.size = odd.size;

} else if (even.size > 0) {

this.head = even.head;

this.tail = even.tail;

this.size = even.size;

}

}

public void kReverse(int k) {

LinkedList prev = null;

while (this.size > 0) {

LinkedList curr = new LinkedList();

if (this.size >= k) {

for (int i = 0; i < k; i++) {

int val = this.getFirst();

this.removeFirst();

curr.addFirst(val);

}

} else {

int sz = this.size;

for (int i = 0; i < sz; i++) {

int val = this.getFirst();

this.removeFirst();

curr.addLast(val);

}

}

if (prev == null) {

prev = curr;

} else {

prev.tail.next = curr.head;

prev.tail = curr.tail;

prev.size += curr.size;

}

}

this.head = prev.head;

this.tail = prev.tail;

this.size = prev.size;

}

private void displayReverseHelper(Node node) {

if (node == null) {

return;

}

displayReverseHelper(node.next);

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

}

public void displayReverse() {

displayReverseHelper(head);

System.out.println();

}

private void reversePRHelper(Node node) {

if (node == tail) {

return;

}

reversePRHelper(node.next);

node.next.next = node;

}

public void reversePR() {

reversePRHelper(head);

Node temp = head;

head = tail;

tail = temp;

tail.next = null;

}

public static int addTwoLists(Node on, Node tn, int pio, int pit, LinkedList res){

if(on == null && tn == null){

return 0;

}

int carry = 0;

int data = 0;

if(pio > pit){

carry = addTwoLists(on.next, tn, pio - 1, pit, res);

data = carry + on.data;

} else if(pio < pit){

carry = addTwoLists(on, tn.next, pio, pit - 1, res);

data = carry + tn.data;

} else {

carry = addTwoLists(on.next, tn.next, pio - 1, pit - 1, res);

data = carry + on.data + tn.data;

}

carry = data / 10;

data = data % 10;

res.addFirst(data);

return carry;

}

public static LinkedList addTwoLists(LinkedList one, LinkedList two) {

LinkedList res = new LinkedList();

int carry = addTwoLists(one.head, two.head, one.size, two.size, res);

if(carry > 0){

res.addFirst(carry);

}

return res;

}

}

public static void main(String[] args) throws Exception {

BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

int n1 = Integer.parseInt(br.readLine());

LinkedList l1 = new LinkedList();

String[] values1 = br.readLine().split(" ");

for (int i = 0; i < n1; i++) {

int d = Integer.parseInt(values1[i]);

l1.addLast(d);

}

int n2 = Integer.parseInt(br.readLine());

LinkedList l2 = new LinkedList();

String[] values2 = br.readLine().split(" ");

for (int i = 0; i < n2; i++) {

int d = Integer.parseInt(values2[i]);

l2.addLast(d);

}

LinkedList sum = LinkedList.addTwoLists(l1, l2);

int a = Integer.parseInt(br.readLine());

int b = Integer.parseInt(br.readLine());

l1.display();

l2.display();

sum.display();

sum.addFirst(a);

sum.addLast(b);

sum.display();

}

}

We highly suggest you watch the solution video for this problem, to see it dry run multiple times.

For more clarity of the question, watch the question video

Play Video

**4. Analysis:**

Time Complexity: O(n)

Since recursion calls are made in this program, therefore its complexity is linear.

Space Complexity: O(n)

The recursion space used in this program is of order n hence the space complexity is linear.