Merge two sorted linked lists

List1: 2 -> 4 -> 5 -> 6 -> 8 -> 9  
List2: 1 -> 3 -> 7  
Merged List:  
1 -> 2 -> 3 -> 4 -> 5 -> 6 -> 7 -> 8 -> 9

## Algorithm/Insights

Create a mergeNext node which points to smaller head.  
Keep traversing the lists, node by node.  
If first list element is smaller, point mergedNext node to first list element and move first list forward by one node else point mergedNext to second list element.

public class LinkedList {

    Node head;

    // Add a new node to the front of the linked list

    public void addToFront(int data) {

        Node n = new Node(data);

        n.next = head;

        head = n;

    }

    // Print list elements

    public void printList() {

        Node tmp = head;

        while(tmp != null) {

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

            tmp = tmp.next;

        }

        System.out.println();

    }

    // Merge 2 sorted lists to form a single sorted list

    public void mergeList(LinkedList list) {

        if(list == null || list.head == null) {

            return;

        }

        if(head == null) {

            head = list.head;

            return;

        }

        Node tmp1 = head;

        Node tmp2 = list.head;

        if(tmp1.data < tmp2.data) {

            head = tmp1;

            tmp1 = tmp1.next;

        } else {

            head = tmp2;

            tmp2 = tmp2.next;

        }

        Node mergedNext = head;

        while(tmp1 != null && tmp2 != null) {

            if(tmp1.data < tmp2.data) {

                mergedNext.next = tmp1;

                tmp1 = tmp1.next;

            } else {

                mergedNext.next = tmp2;

                tmp2 = tmp2.next;

            }

            mergedNext = mergedNext.next;

        }

        if(tmp1 != null) {

            mergedNext.next = tmp1;

        } else {

            mergedNext.next = tmp2;

        }

    }

    public static void main(String[] args) {

        LinkedList list1 = new LinkedList();

        list1.addToFront(9);

        list1.addToFront(8);

        list1.addToFront(6);

        list1.addToFront(5);

        list1.addToFront(4);

        list1.addToFront(2);

        LinkedList list2 = new LinkedList();

        list2.addToFront(7);

        list2.addToFront(3);

        list2.addToFront(1);

        list1.mergeList(list2);

        System.out.println("Merged List:");

        list1.printList();

    }

}

class Node {

    int data;

    Node next;

    public Node(int data) {

        this.data = data;

    }

# Reverse a Linked List - Recursive

Given a linked list having n nodes. Reverse the list using recursive approach.

Start with node curr as head.  
1. If curr is null, return.  
2. If curr's next element is null, this means it is the last node, so make this as head because the last node will be the head of reversed list. Return.  
3. Recursively traverse the list.   
4. Set curr->next->next to curr.  
5. Set curr->next to null

public class ReverseLinkedListRecursive {

    private Node head;

    public Node getHead() {

        return head;

    }

    public void setHead(Node head) {

        this.head = head;

    }

    public void reverseLinkedListRecursive() {

        reverseLinkedListRecursive(head);

    }

    private void reverseLinkedListRecursive(Node curr) {

        if (curr == null) {

            return;

        }

        if (curr.getNext() == null) {

            this.head = curr;

            return;

        }

        reverseLinkedListRecursive(curr.getNext());

        curr.getNext().setNext(curr);

        curr.setNext(null);

    }

    /\*

     \* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

     \* Following methods are for testing the solution

     \* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

     \*/

    public static void main(String[] args) {

        ReverseLinkedListRecursive list = new ReverseLinkedListRecursive();

        list.createTestList(5);

        list.printlist();

        list.reverseLinkedListRecursive();

        list.printlist();

    }

    /\*

     \* Create a test list having n nodes from 1 to n

     \*/

    public void createTestList(int n) {

        if(n < 1)

            return;

        int i = 1;

        Node temp = null;

        while(i <= n) {

            Node node = new Node(i);

            if(head == null) {

                head = node;

                temp = head;

            } else {

                temp.setNext(node);

                temp = node;

            }

            i++;

        }

    }

    /\*

     \* Print the list

     \*/

    public void printlist() {

        Node temp = head;

        while(temp != null) {

            System.out.print(temp.getData() + " -> ");

            temp = temp.getNext();

        }

        System.out.println("X");

    }

    /\*\*

     \* Defines a linked list node class

     \* @author Saurabh

     \*

     \*/

    class Node {

        private int data;

        private Node next;

        public int getData() {

            return data;

        }

        public void setData(int data) {

            this.data = data;

        }

        public Node getNext() {

            return next;

        }

        public void setNext(Node next) {

            this.next = next;

        }

        public Node(int data) {

            super();

            this.data = data;

        }

    }

# Reverse a Linked List - Iterative

Given a linked list having n nodes. Reverse the list using iterative approach.

Keep 3 pointers - prev (previous node), curr (current node) and nxt (next node).  
1. Initialize prev = null, curr = null, nxt = head.   
2. Set curr = nxt.  
3. Move nxt to next node pointer.  
4. Set curr next to prev.  
5. Set prev to curr  
6. Repeat steps 2-5 till next is not null.  
7. Set curr as head pointer of the list.

/\*

     \* Create a test list having n nodes from 1 to n

     \*/

    public void createTestList(int n) {

        if(n < 1)

            return;

        int i = 1;

        Node temp = null;

        while(i <= n) {

            Node node = new Node(i);

            if(head == null) {

                head = node;

                temp = head;

            } else {

                temp.setNext(node);

                temp = node;

            }

            i++;

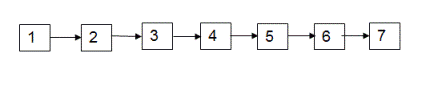
        }

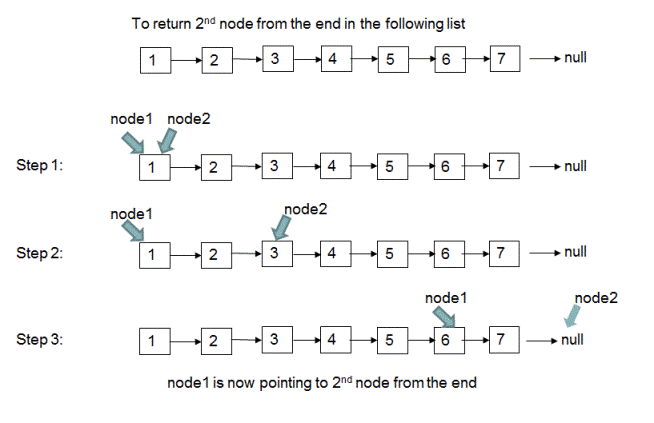
    }

# Reverse every alternate k nodes of a Linked List

Given a linked list and a number k, reverse every alternate k nodes of the list.  
Example:  
List: 1 - 2 - 3 - 4 - 5 - 6 - 7 - 8 - 9 - 10 - null  
k = 4  
Output: 4 - 3 - 2 - 1 - 5 - 6 - 7 - 8 - 10 - 9 – null

# Find nth Node from the end of a Linked List

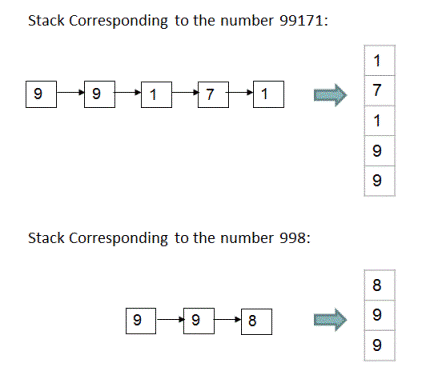
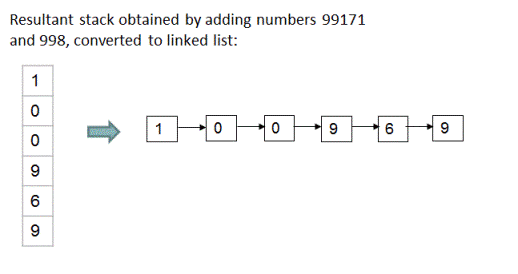
Given a linked list, find 'n'th node from the end for a given value of n (n > 0). For example, for the following linked list  
   
if n = 2, we have to find 2nd node from the end which is node 6.  
if n = 1, we have to find 1st node from the end which is node 7.

**Example:**  


# Sum of Two Linked Lists using Stacks

iven two numbers which are represented using linked lists as shown below. Your program should return the reference to a new linked list which stores the sum of given two numbers.   
  
Numbers are represented as following:  
Number 99971, corresponding linked list: 9->9->9->7->1  
Number 998,   corresponding linked list: 9->9->8

The output returned by the program for above two linked lists as the input should be the linked list 1->0->0->9->6->9 which represents number 100969 which is sum of numbers 99971 and 998.

**Idea:** Because the unit digits of numbers are placed at the end of the linked lists which represent them, to add given two numbers, we need to start adding the values of the nodes starting from the end nodes of linked lists. For example, to add numbers 99,171(9->9->9->7->1) and 998(9->9->8), we start by adding node with value '1' of the first list to the node with value '8' of the second list. To enforce this reverse ordering, we can push the given two lists to two different stacks thereby placing the unit digits at the top of the stacks and then starting the addition process by popping digits from the stacks one by one. Also since we need to return the resultant number represented in the form of a linked list(which has unit digit at the end), we push the addition result onto a stack and create the linked list out of this stack. For adding two numbers 99,171 and 998, the stacks will look like following after pushing the two linked lists to stacks.  
          
And the result of addition of these two numbers would again be pushed to stack digit by digit. This resultant stack then would be converted to the linked list.  
  
  
**Algorithm:** The formal steps of this algorithm are as following:  
1. Create stack 's1' by pushing all node values of the first linked list to a stack.  
2. Create stack 's2' by pushing all node values of the second linked list to a stack.  
3. Create an empty stack 's3' to store the result of addition.  
4. Initialize sum and carry to 0.  
5. Pop the top element from stack 's1'. Let this top element be 'value1'.  
6. Pop the top element from stack 's2'. Let this top element be 'value2'.  
7. Make 'sum' = (value1 + value2 + carry)%10 and push this 'sum' to stack 's3' and make 'carry' = (value1 + value2 + carry)/10.  
8. Repeat steps 5-7 till one of the stacks become empty. If both stacks are of same size, then both of the stacks would become empty at the same time.   
9. If stack 's1' has elements left in it then -   
    a. Pop the top element from stack 's1'. Let this top element be 'value1'.   
    b. Make 'sum' = (value1 + carry)%10 and push this 'sum' to stack 's3' and make 'carry' = (value1 + carry)/10.  
    c. Repeat steps 9a and 9b until stack 's1' is not empty.  
10. Similarly, if stack 's2' has elements left in it then -   
    a. Pop the top element from stack 's2'. Let this top element be 'value2'.   
    b. Make 'sum' = (value2 + carry)%10 and push this 'sum' to stack 's3' and make 'carry' = (value2 + carry)/10.  
    c. Repeat steps 10a and 10b until stack 's2' is not empty.  
11. After all the above steps are executed, if carry is greater than 0, push it to the resultant stack 's3'.  
12. Create an empty linked list 'result'. Now pop elements one by one from the stack 's3', and keep appending them to the 'result' linked list until stack 's3' is not empty. Return the output as 'result' linked list.    
      
Please check out the function 'addLists(ListNode node1, ListNode node2)' from code snippet below for implementation details. The time and space complexity of this approach is O(n).

# LRU Cache Implementation

## Algorithm/Insights

For implementing an LRU cache, we can use a doubly linked list and a hash map.  
Doubly Linked List - List of pages with most recently used page at the start of the list. So, as more pages are added to the list, least recently used pages are moved to the end of the list with page at tail being the least recently used page in the list.  
Hash Map (key: page number, value: page) - For O(1) access to pages in cache  
  
When a page is accessed, there can be 2 cases:  
1. Page is present in the cache - If the page is already present in the cache, we move the page to the start of the list.  
2. Page is not present in the cache - If the page is not present in the cache, we add the page to the list.   
How to add a page to the list:  
   a. If the cache is not full, add the new page to the start of the list.  
   b. If the cache is full, remove the last node of the linked list and move the new page to the start of the list.  
  
Java code is provided in code snippet section.  
Please refer to Algorithm Visualization section for understanding how the algorithm works for different test cases.

public class LRUCache {

    private DoublyLinkedList pageList;

    private Map<Integer, Node> pageMap;

    private final int cacheSize;

    public LRUCache(int cacheSize) {

        this.cacheSize = cacheSize;

        pageList = new DoublyLinkedList(cacheSize);

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

    }

    public void accessPage(int pageNumber) {

        Node pageNode = null;

        if(pageMap.containsKey(pageNumber)) {

            // If page is present in the cache, move the page to the start of list

            pageNode = pageMap.get(pageNumber);

            pageList.movePageToHead(pageNode);

        } else {

            // If the page is not present in the cache, add the page to the cache

            if(pageList.getCurrSize() == pageList.getSize()) {

                // If cache is full, we will remove the tail from the cache pageList

                // Remove it from map too

                pageMap.remove(pageList.getTail().getPageNumber());

            }

            pageNode = pageList.addPageToList(pageNumber);

            pageMap.put(pageNumber, pageNode);

        }

    }

    public void printCacheState() {

        pageList.printList();

        System.out.println();

    }

    public static void main(String[] args) {

        int cacheSize = 4;

        LRUCache cache = new LRUCache(cacheSize);

        cache.accessPage(4);

        cache.printCacheState();

        cache.accessPage(2);

        cache.printCacheState();

        cache.accessPage(1);

        cache.printCacheState();

        cache.accessPage(1);

        cache.printCacheState();

        cache.accessPage(4);

        cache.printCacheState();

        cache.accessPage(3);

        cache.printCacheState();

        cache.accessPage(7);

        cache.printCacheState();

        cache.accessPage(8);

        cache.printCacheState();

        cache.accessPage(3);

        cache.printCacheState();

    }

}

class DoublyLinkedList {

    private final int size;

    private int currSize;

    private Node head;

    private Node tail;

    public DoublyLinkedList(int size) {

        this.size = size;

        currSize = 0;

    }

    public Node getTail() {

        return tail;

    }

    public void printList() {

        if(head == null) {

            return;

        }

        Node tmp = head;

        while(tmp != null) {

            System.out.print(tmp);

            tmp = tmp.getNext();

        }

    }

    public Node addPageToList(int pageNumber) {

        Node pageNode = new Node(pageNumber);

        if(head == null) {

            head = pageNode;

            tail = pageNode;

            currSize = 1;

            return pageNode;

        } else if(currSize < size) {

            currSize++;

        } else {

            tail = tail.getPrev();

            tail.setNext(null);

        }

        pageNode.setNext(head);

        head.setPrev(pageNode);

        head = pageNode;

        return pageNode;

    }

    public void movePageToHead(Node pageNode) {

        if(pageNode == null || pageNode == head) {

            return;

        }

        if(pageNode == tail) {

            tail = tail.getPrev();

            tail.setNext(null);

        }

        Node prev = pageNode.getPrev();

        Node next = pageNode.getNext();

        prev.setNext(next);

        if(next != null) {

            next.setPrev(prev);

        }

        pageNode.setPrev(null);

        pageNode.setNext(head);

        head.setPrev(pageNode);

        head = pageNode;

    }

    public int getCurrSize() {

        return currSize;

    }

    public void setCurrSize(int currSize) {

        this.currSize = currSize;

    }

    public Node getHead() {

        return head;

    }

    public void setHead(Node head) {

        this.head = head;

    }

    public int getSize() {

        return size;

    }

}

class Node {

    private int pageNumber;

    private Node prev;

    private Node next;

    public Node(int pageNumber) {

        this.pageNumber = pageNumber;

    }

    public int getPageNumber() {

        return pageNumber;

    }

    public void setPageNumber(int data) {

        this.pageNumber = data;

    }

    public Node getPrev() {

        return prev;

    }

    public void setPrev(Node prev) {

        this.prev = prev;

    }

    public Node getNext() {

        return next;

    }

    public void setNext(Node next) {

        this.next = next;

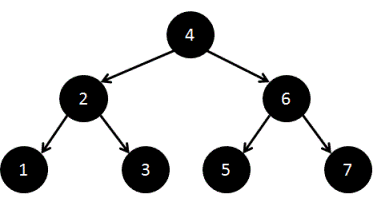
    }

    public String toString() {

        return pageNumber + "  ";

# Detect a loop in a linked list and find the node where the loop starts

# Convert a sorted Doubly Linked List to Balanced Binary Search Tree

Given a doubly linked list in sorted order with previous and next nodes. Convert the doubly linked list to a binary search tree with left as previous node and right as next node.   
Consider the list below:  
  
The list should be converted to following BST:  


We recursively traverse to the leaves and then create the tree upwards from the leaves to the root.  
Step 1. Calculate the length of the linked list.  
Step 2. Recursively create left sub tree from first half nodes.  
Step 3. Make middle node as the root and node returned from previous call (Step 2) as left child of the root.  
Step 4. Move head to next node.  
Step 5. Recursively create the right sub tree from second half nodes.  
Step 6. Return root.

public class DllToBst {

    private ListNode head = null;

    private ListNode tail = null;

    public void addToList(int data) {

        ListNode n = new ListNode(data);

        if (head == null) {

            head = n;

            tail = n;

        } else {

            // Add to end of list

            n.prev = tail;

            tail.next = n;

            tail = n;

        }

    }

    /\*\*

     \* Convert DLL to BST:

     \* prev node will be right child

     \* next node will be left child

     \*/

    public void convertDllToBST() {

        int len = getListLength();

        // head is root of BST, tail is null.

        head = convertDllToBST(len);

        tail = null;

    }

    private int getListLength() {

        int len = 0;

        ListNode curr = head;

        while (curr != null) {

            len++;

            curr = curr.next;

        }

        return len;

    }

    private ListNode convertDllToBST(int len) {

        if (len == 0) {

            return null;

        }

        ListNode left = convertDllToBST(len / 2);

        ListNode root = head;

        root.prev = left;

        head = head.next;

        ListNode right = convertDllToBST(len - (len / 2) - 1);

        root.next = right;

        return root;

    }

    public void printInorderOrderTraversal() {

        printInorderOrderTraversalHelper(head);

    }

    private void printInorderOrderTraversalHelper(ListNode root) {

        if (root == null) {

            return;

        }

        printInorderOrderTraversalHelper(root.prev);

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

        printInorderOrderTraversalHelper(root.next);

    }

    public static void main(String args[]) {

        DllToBst dll = new DllToBst();

        for (int i = 1; i < 8; i++) {

            dll.addToList(i);

        }

        dll.convertDllToBST();

        dll.printInorderOrderTraversal();

    }

}

class ListNode {

    int data;

    ListNode prev;

    ListNode next;

    public ListNode(int data) {

        this.data = data;

    }

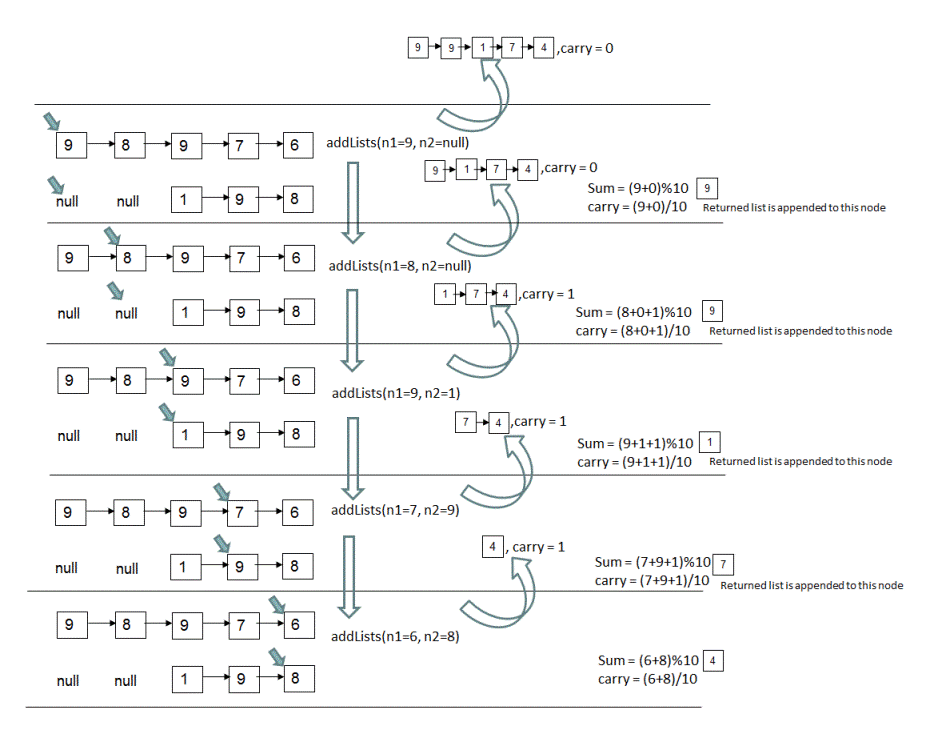
# Convert a binary tree to doubly linked list

Given a binary tree, convert it into a doubly linked list as described:  
1. We do not have to create a new linked list data structure. We have to convert the tree to a doubly linked list.  
2. The doubly linked list should be created such that nodes follow inorder traversal of the binary tree.  
3. Left node of the binary tree should be converted to the previous node of the doubly linked list.  
4. Right node of the binary tree should be converted to the next node of the doubly linked list.  
5. Left most node of the binary tree should be the head of the linked list.  
  
Example:  
Tree:  
                    1  
            2                3  
        4        5        6        7  
  
Output: 4 <-> 2 <-> 5 <-> 1 <-> 6 <-> 3 <-> 7

Process left subtree first:  
    1. Move to the right most node of left subtree to get the inorder predecessor of the root, lets call is A.  
    2. Now create doubly linked list pointers:  
       a. Set the A.right to the root.  
       b. Set root.left to A.  
Process right subtree:  
    1. Move to the left most node of right subtree to get the inorder successor of the root, lets call is B.  
    2. Now create doubly linked list pointers:  
       a. Set the B.left to the root.  
       b. Set root.right to B.  
Please go through algorithm visualization section and java code in code snippet section.

# Sum of Two Linked Lists using Recursion | Set 1

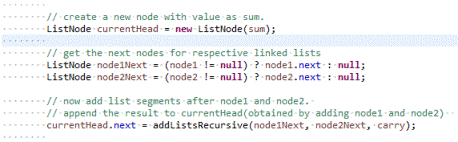
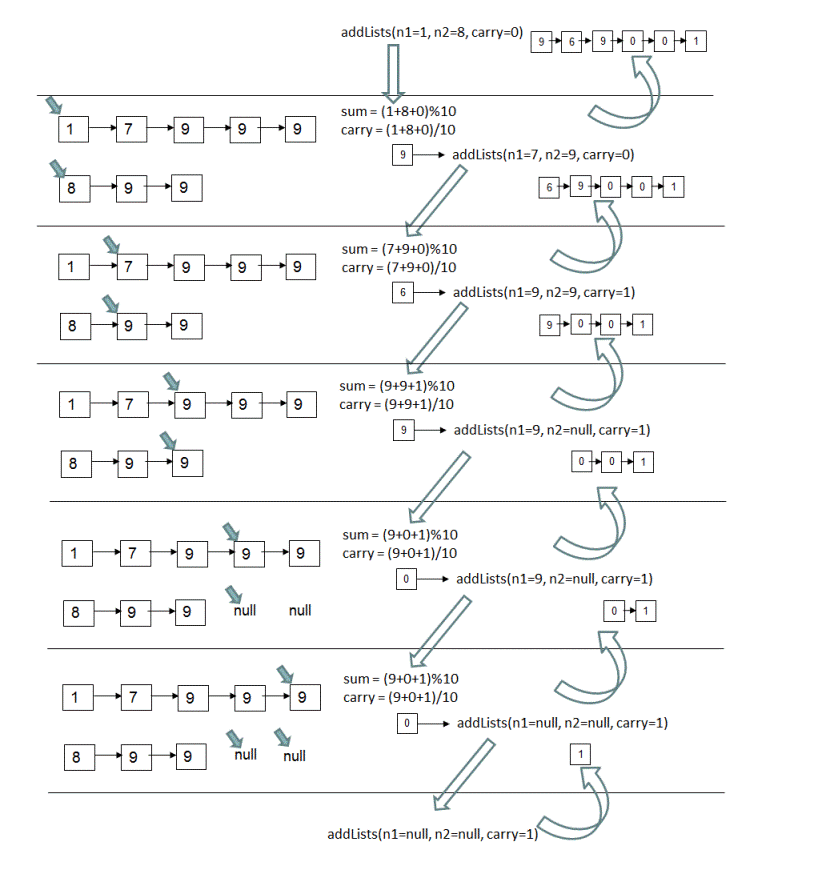
Given two numbers which are represented using linked lists as shown below. Return the reference to a new linked list which stores the sum of given two numbers. You are not allowed to make use of explicit extra space except temporary variables.  
  
Numbers are represented as following:  
Number 98976, corresponding linked list: 9->8->9->7->6->null  
Number 198,   corresponding linked list: 1->9->8->null  
  
The output returned by the program for above two linked lists as input should be the linked list 9->9->1->7->4->null

**Java implementation details:** In the above recursive algorithm, we are assuming that the function addLists(ListNode node1, ListNode node2) returns link list and carry obtained by adding lists starting from 'node1' and 'node2'. But Java does not allow two objects to be returned. As a work around for this limitation, we pass int array carry[] in the function as an argument and modify carry[0] in each function call which will be then used by the calling function.  
  
The logical flow of the recursive calls for adding lists 9->8->9->7->6->null and 1->9->8->null is shown below:  


# Sum of Two Linked Lists using Recursion | Set 2

In [this post](http://www.ideserve.co.in/learn/sum-of-two-linked-lists-using-recursion-set-1) we have covered the recursive solution for finding out sum of numbers represented by linked lists. In that question though, a number was represented in such a way that its unit digit was placed at the end of the linked list. For example, number 98734 would be represented as 9->8->7->3->4.   
  
In this problem, a number is represented in reverse fashion, that is its unit digit is placed at the beginning of the linked list. For example, number 98734 is represented as 4->3->7->8->9. Given this representation, write a program that takes input as two linked lists(which represent two numbers) and returns output as the linked list which is sum of two numbers represented by input linked lists.   
  
Example:   
Input: 1->7->9->9->9->null(number 99971), 8->9->9->null(number 998).  
Output: 9->6->9->0->0->1->null(number 100969)

## Algorithm/Insights

This problem could be solved using either recursive or iterative method. We describe the algorithm for recursive method below along with its pictorial representation. Iterative algorithm is very similar and it is also implemented using function addListsIterative(ListNode node1, ListNode node2) in the code snippet below. You can check the code snippet for details.  
  
**Recursive Algorithm:** Since first nodes of linked lists point to unit digit of respective numbers they are representing, we think this algorithm is relatively easy to understand than [this algorithm](http://www.ideserve.co.in/learn/sum-of-two-linked-lists-using-recursion-set-1) where last nodes of lists point to unit digits of respective numbers.  
  
If function addListsRecursive(ListNode node1, ListNode node2, int carry) implements the algorithm then the steps of this function are as following -   
1. First call to addListsRecursive(ListNode node1, ListNode node2, int carry) is made with 'node1' as first node of first input list, 'node2' as first node of second input list and carry as 0.  
2. **Base case:**If 'node1' and 'node2' both are null then we check if 'carry' is greater than 0. If 'carry' is greater than 0, then we create a node with value as 'carry' and return this node. But if 'carry' is 0 then we return null from this function.  
**Recursive Steps:**  
3. If 'node1' is not null, we assign value of 'node1' to 'value1' otherwise we assign 0 to 'value1'. Similarly, if 'node2' is not null, we assign value of 'node2' to 'value2' otherwise we assign 0 to 'value2'.   
4. Using 'value1' and 'value2' from above step and using 'carry' from the argument to this function, we compute 'sum' and 'carry' as: sum = (value1+value2+carry)%10, carry = (value1+value2+carry)/10.  
5. Now we create a new node say 'currentHead' with its value as 'sum'(computed in step #4).  
6. To add the remaining segments of linked lists which would be segments after 'node1' and 'node2', we make a recursive call as : addListsRecursive(node1Next, node2Next, carry) where 'node1Next' and 'node2Next' are the next nodes of 'node1' and 'node2' respectively in their corresponding lists and 'carry' is computed in step #4. The resultant list returned by this recursive call is appended to 'currentHead' created in step #5. Please note that next node of a null node is considered as null node. This step is implemented as following:  
  
7. Since 'currentHead' now points to the first node of the resultant list obtained by adding lists starting from 'node1' and 'node2' onwards, we return 'currentHead' from this function to the calling function.  
  
For more clarity, you may want to refer to following pictorial representation of this algorithm for input lists as 1->7->9->9->9->null(number 99971) and 8->9->9->null(number 998).  


# Find intersection of two Linked Lists

Two linked lists A and B are joined on a particular node, called the point of intersection of the linked lists. Find the point of intersection, i.e. the first node after which both lists have same nodes.

Naive Solution:  
Keep a node in List A as fixed and iterate over all nodes in List B to check if there is a node in List B same as the fixed node from List A.  
1. Take two loops - outer loop for iterating over list A. Inner loop for list B.  
2. In inner loop, check if a node matches current node of A.  
3. If found, return the node, else move to next node.  
4. If not found, return null.  
  
Time Complexity: O(n\*m)  
Space Complexity: O(1)  
  
Solution using a Hash:  
1. Iterate over first list and put the nodes in a hash set.  
2. Iterate over other list and find the first node that is present in the hash set. This node is the intersection of the 2 lists.  
  
Time Complexity: O(n+m)  
Space Complexity: O(n)

# Find intersection of two Linked Lists - O(m + n) Time Complexity and O(1) Space Complexity

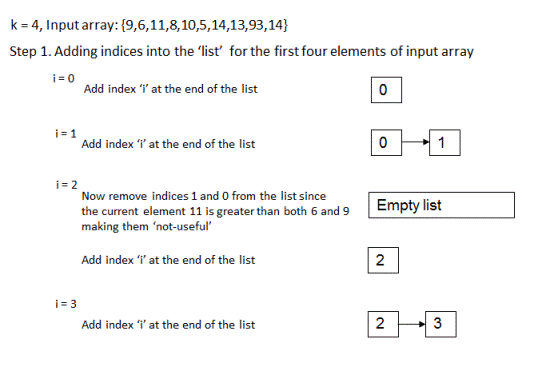
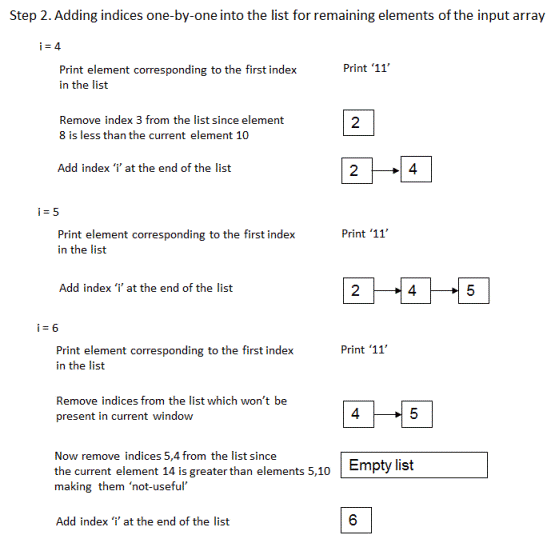
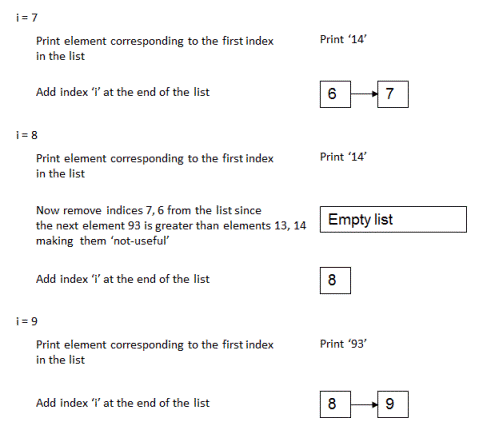
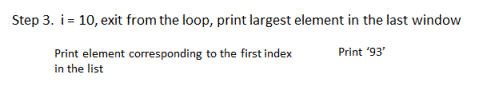
Two linked lists list1 and list2 are joined a particular node, called the point of intersection of the linked lists. Find the point of intersection, i.e. the first node after which both lists have same nodes.   
Desired order is O(m + n) Time Complexity and O(1) Space Complexity

## Algorithm/Insights

1. Find length of list1 - use a tmp1 node starting from head of list1 and move till last node.  
2. Find length of list2 - use a tmp2 node starting from head of list2 and move till last node.  
3. If tmp1 and tmp2 are different, it means that linked lists are non-intersecting. Return null.  
   Example: list1: 1-2-3-4 , list2: 5-6-7-8, last nodes are different.  
4. Else set variables diff, tmp1 and tmp2 as:  
   a). tmp1 (a list node) to head node of larger list.  
   b). tmp2 (a list node) to head node of smaller list.  
   c). diff (an integer) to difference of lengths of larger to smaller lists i.e. absolute difference of the lengths.  
5. Move forward tmp1 by diff number of nodes.  
6. Now lists starting from tmp1 and tmp2 have same number of nodes and intersect at a particular node. Therefore, both tmp1 and tmp2 are equidistant from the intersection node.  
7. Starting from tmp1 and tmp2 simultaneously, move node by node till a common node is reached. This node is the intersection of the 2 lists.

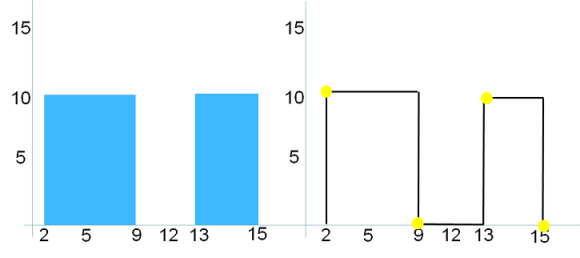
# Find maximum element from each sub-array of size 'k'| Set 2

If you are given an integer array and an integer 'k' as input, write a program to print elements with maximum values from each possible sub-array (of given input array) of size 'k'. If the given input array is {9,6,11,8,10,5,14,13,93,14} and for k = 4, output should be 11,11,11,14,14,93,93. Please observe that 11 is the largest element in the first, second and third sub-arrays - {9,6,11,8}, {6,11,8,10} and {11,8,10,5}; 14 is the largest element for fourth and fifth sub-array and 93 is the largest element for remaining sub-arrays.

Now let's try to understand the steps of the algorithm.  
1. For first 'k' elements of the input array that is for i = 0 up to i = k-1 :   
(a). Before adding index 'i' into the 'list', we remove all indices 'j's from the rear end of the 'list' for which array[j] is less than array[i]. This is because the elements corresponding to these indices are never going to the largest elements for this sub-array of first 'k' elements as well as for any subsequent sub-array. Hereafter, we call such elements as 'not-useful' elements. We make use of the following 'while loop' for this step -   
    while (!list.isEmpty() && array[list.getLast()] < array[i])  
    {  
        list.removeLast();  
    }  
(b). Now we add index 'i' at the end of the 'list'.  
  
After execution of step #1a and #1b, 'list' contains indices corresponding to useful elements only and the first index in the list corresponds to the largest element from the first sub-array(window) of 'k' elements. In the below steps as well, we will maintain the same state of the list where the first index in the list corresponds to the largest element from the current window of 'k' elements and the 'list' contains indices for useful elements only.  
  
2. Now for remaining elements of the input array that is for i = k up to i = n-1 (n: length of the input array) -   
(a). We print the element corresponding to the first index in the 'list'. This element would be the largest element in the last seen window of 'k' elements.  
(b). Now we remove all indices from the front-end of the 'list' which won't be included in the current window of size 'k'. Note that for this current window of size 'k', index of the last element would be 'i', index of the second last element would be 'i-1' and the index of the first element would be 'i-k+1'. Any indices present in the list which are less than 'i-k+1' won't be included in current window. For this step, we use the following 'while loop' -   
    while (!list.isEmpty() && (list.getFirst() < (i-k+1)))  
    {  
        list.removeFirst();  
    }  
(c). Now very similar to step #1a, we remove all indices corresponding to 'not-useful' elements starting from the rear-end of the 'list'. This helps to maintain the indices in the order such that corresponding array elements are sorted in the decreasing order.    
(d). We add index 'i' at the end of the 'list'.  
  
3. After execution of step #2, the first index in the list now corresponds to the largest element in the last window of size 'k'. We print the element corresponding to this index which marks the completion of this algorithm.  
     
Here is a step by step illustration of this algorithm for the input array as {9,6,11,8,10,5,14,13,93,14} and for k = 4,  
  
  
  
  
     
The time complexity of this algorithm is O(n) where 'n' is the length of the input array. Note that in this algorithm every element is compared at most twice for remove operation from the rear-end and every element on an average is compared only once for remove operation from the front-end of the 'list'. Therefore, for every element of the array there are constant number of operations that are performed on it making time complexity as O(n). The extra space used by this algorithm is O(k).

# Find maximum element from each sub-array of size 'k'| Set 2

If you are given an integer array and an integer 'k' as input, write a program to print elements with maximum values from each possible sub-array (of given input array) of size 'k'. If the given input array is {9,6,11,8,10,5,14,13,93,14} and for k = 4, output should be 11,11,11,14,14,93,93. Please observe that 11 is the largest element in the first, second and third sub-arrays - {9,6,11,8}, {6,11,8,10} and {11,8,10,5}; 14 is the largest element for fourth and fifth sub-array and 93 is the largest element for remaining sub-arrays.

  
  
If input is  [[2,9,10], [3,6,15], [5,12,12], [13,16,10], [13,16,10], [15,17,5]]  then the output should be [[2,10], [3,15], [6,12], [12,0], [13,10], [16,5], [17,0]] as shown below.   
