**1. Problem Discussion :**

You are given a partially written LinkedList class. You are required to complete the body of fold function. The function is expected to place the last element after the 1st element, 2nd last element after 2nd element, and so on. For more insight check the example

Example 1

1->2->3->4->5 will fold as 1->5->2->4->3

Example 2

1->2->3->4->5->6 1->6->2->5->3->4

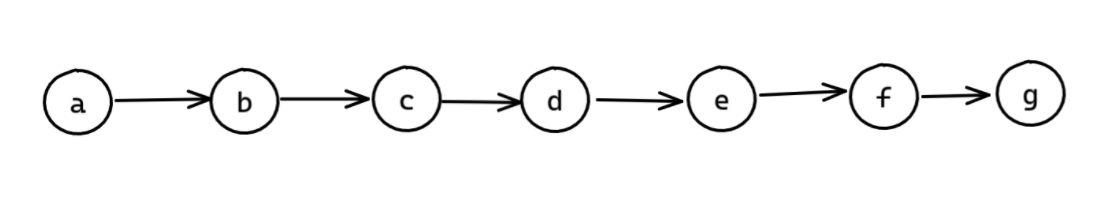
**2. Approach :**

● This is again a very similar approach. Always remember when we will need to access the first element and the last element, the second element, and the second last element, and so on, we can apply this trick.

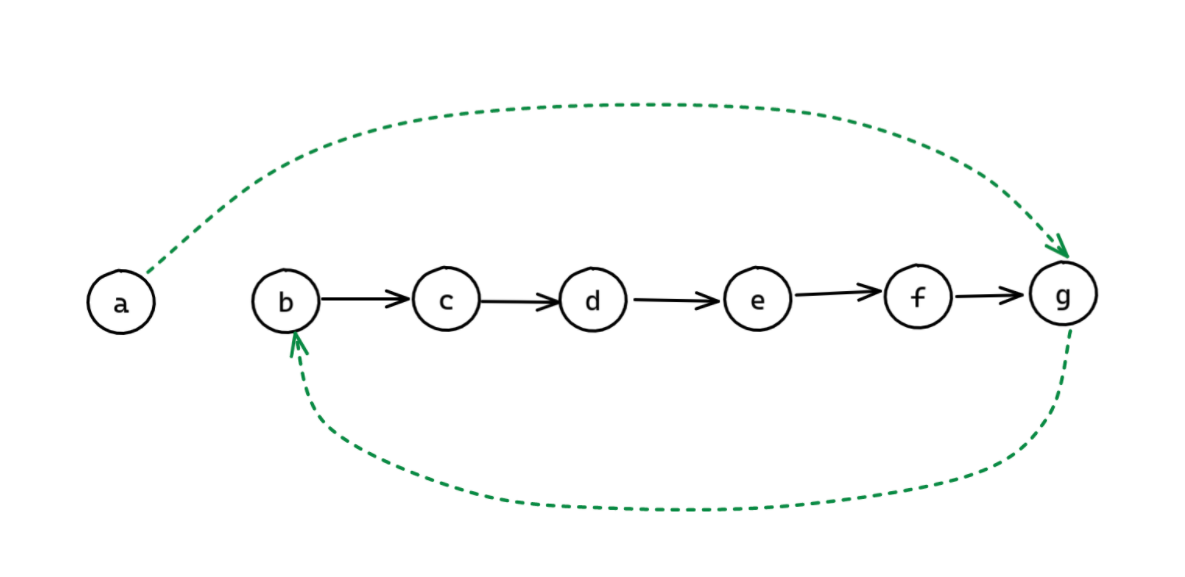
● et's explain briefly what we will do, we will have a global left pointer and we will have a right pointer which will be passed through the stack. How? Bypassing it as a param to a recursive call. And as the recursive function call gets popped from the runtime stack, we will move the left pointer to the next value. This way if the left points to the i-th node the right will point to the n-i-1 th node.

**3. Procedure :**

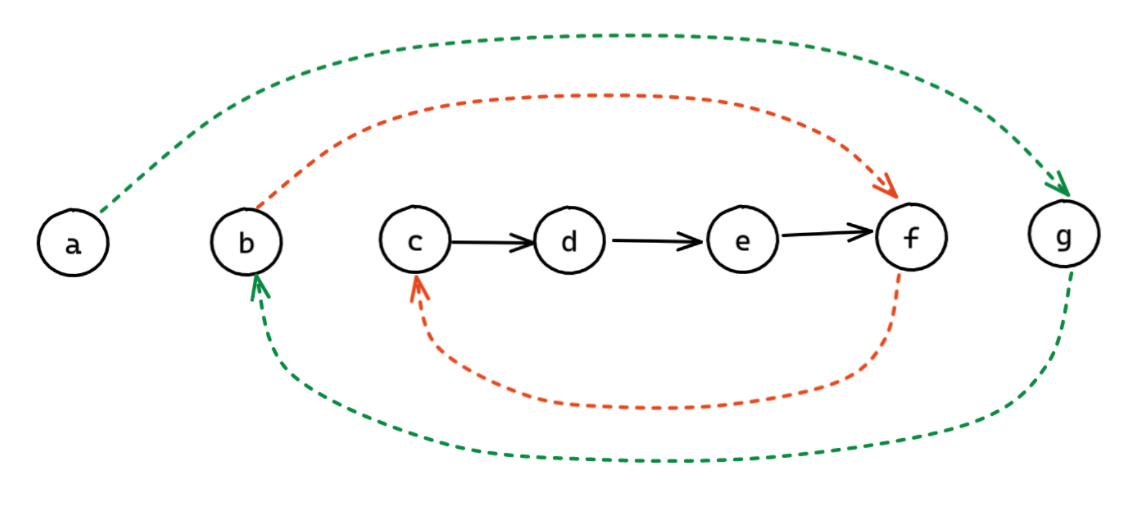
Now, we know how to access the left and right nodes, how do we fold it. So, let's do it this way. Imagine the linked list looks like this:

****

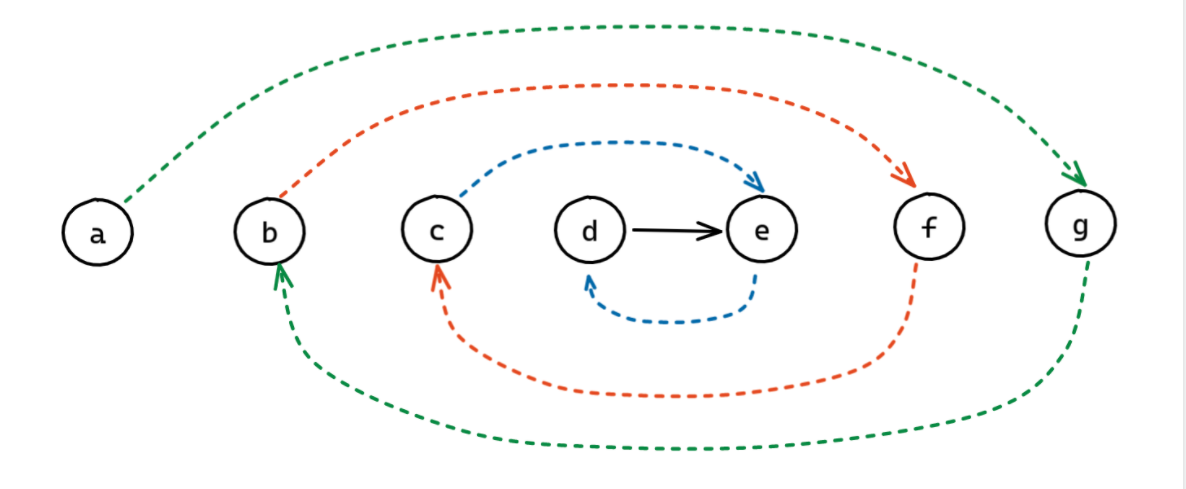
● What we can do is make a's next g and g's next as b.

****

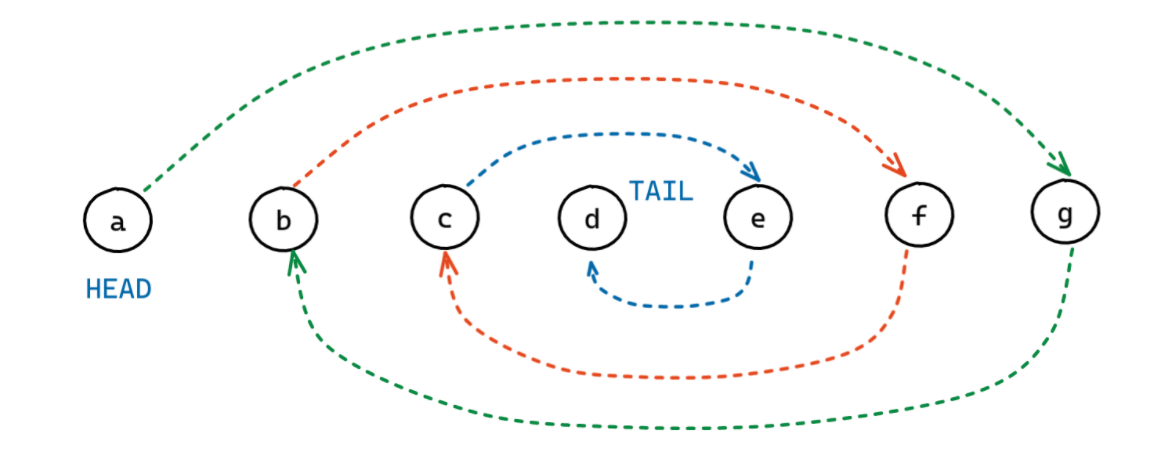
● Maybe it's not clear to you right now, but hand on!!. Then now our left is at b and our right is at f. Now, b's next becomes f and f's next becomes c. Then it will look like:

****

● Now finally the left is at d and the right is at e. Then we will make c's next as e and e's next as d.

****

● Now, I will make a as the head and d as the tail

****

● So, now if we traverse the linked list the ordering will be a->g->b->f->c->e->d Which is the fold of the original one.

● Now, remember we will make a foldHelper function which will be called recursively and all the operations to be done in the post recursive function call. Before performing any operation, the runtime stack will be filled with all the elements, and then we will return from the function call i.e pop the item at the top of the stack and at the same time increment the left to point to its next. Also, while changing the pointers remember to save the left next in a temporary variable, cause we will need it to increment the left pointer.

If you faced any difficulty and want to look at the dry run watch the solution video

For more clarity of the question, watch the question video

Play Video

**4. 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;

}

Node fleft;

private void foldHelper(Node right, int floor) {

if (right == null) {

return;

}

foldHelper(right.next, floor + 1);

if (floor > size / 2) {

Node temp = fleft.next;

fleft.next = right;

right.next = temp;

fleft = temp;

} else if (floor == size / 2) {

tail = right;

tail.next = null;

}

}

public void fold() {

fleft = head;

foldHelper(head, 0);

}

}

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

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

l1.display();

l1.fold();

l1.display();

l1.addFirst(a);

l1.addLast(b);

l1.display();

}

}

**5. Analysis:**

Time Complexity:

● O(n) Here we are again simply visiting the nodes recursively there are n nodes. And each visit takes constant time, hence O(n) time complexity overall.

Space Complexity:

● O(n) Even though we are not using any auxiliary space, we are using the run-time stack n times to store the params of the recursive function. So it will be O(n) as well.

a