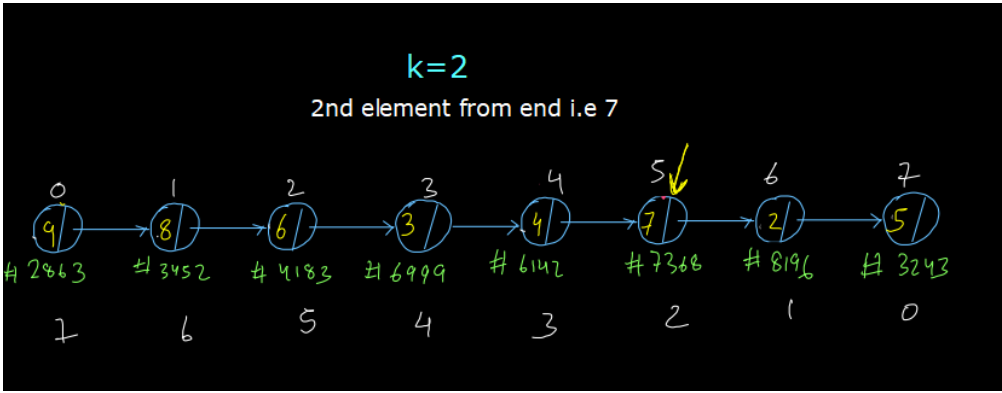
**1. Problem Discussion :**

● We have a linked list of integers and we are also given an integer "k". We have to find the "kth" node from the end of the linked list. The index of the first node is 0. (Have a look at fig-1)

****

● For instance, we are given k=2. So, we had to return the second element from the end of the linked list (i.e. if the last element has index=0, we have to return the element with index=2). We have to return 7 as it is the second element from the end of the linked list. Also, there are 3 constraints.

● You have to solve this question using iteration

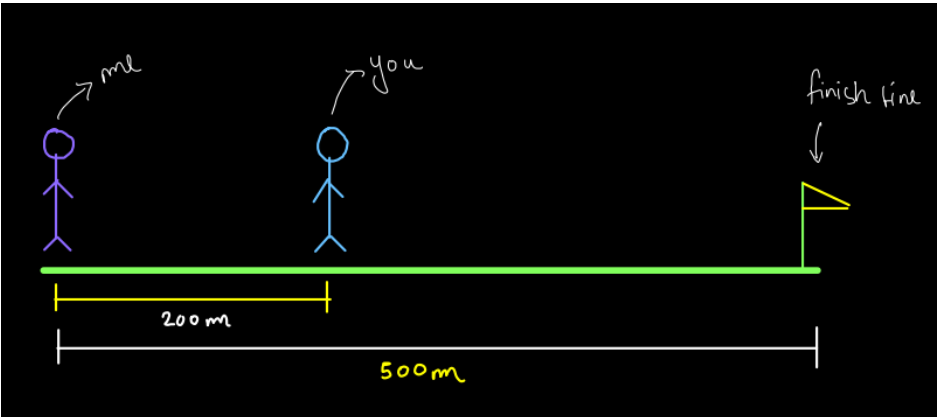
● You do not have to use the size property of the linked list directly or indirectly meaning you cannot use the available size function or size variable and you are not allowed to calculate the size of the linked list either. You have to solve this problem without knowing the size of the linked list.

**2. Approach :**

● So, did you try to solve this problem on your own? There is this trick here in the constraints that you cannot solve the problem by using the size property. Otherwise, it would have been a piece of cake, right? So, the solution to this problem is called the Two Pointer Approach and it is not only used in this problem, but it is used in many problems. Let us try to understand the basis of this two pointer approach first in a fun way.

● The Idea Behind The Two Pointer Approach:

● Let us consider a racing track (500m long). Let us say there are two people. They are you and me. So, my friend, you and I are on a racing track and the race has not started yet. You are already standing at the 200m spot while I am at the starting position i.e. the 0m spot. (Have a look at fig-4)

****

● Now, let us assume that we run at the exact same speed. So, when the race starts, both of us start running and we run at the exact same speed, say 20m/s. So, what do you think? Who is going to be the winner? Of course, it is going to be you (and believe me I am not jealous).

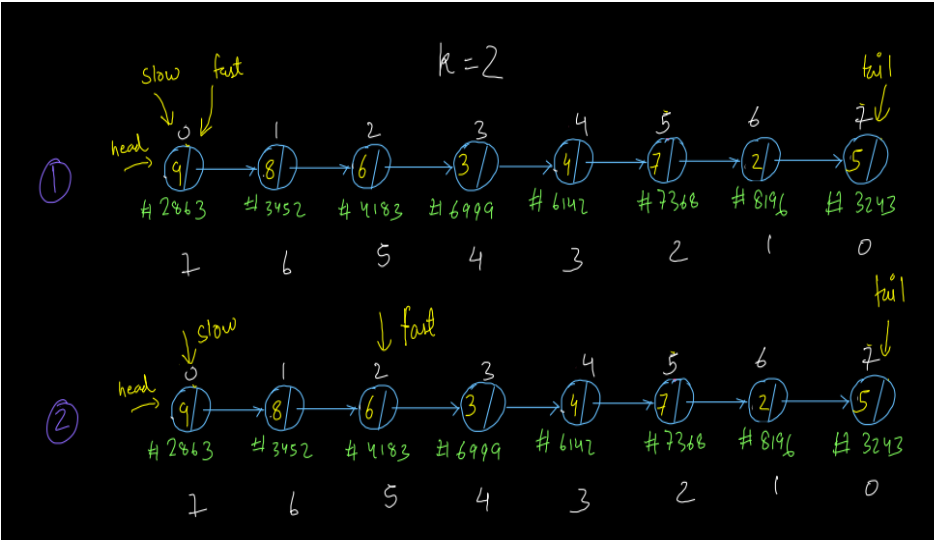
● Can you think why you would win if we were running at the exact same speed? It's obvious that you started from the 200m spot and I was at the 0m spot.

● What do you think will be the distance between us when you finish the race? Well, it is still going to be 200m because we ran at the same speed. So, the distance between us remained constant throughout the race as we ran with the same speed.

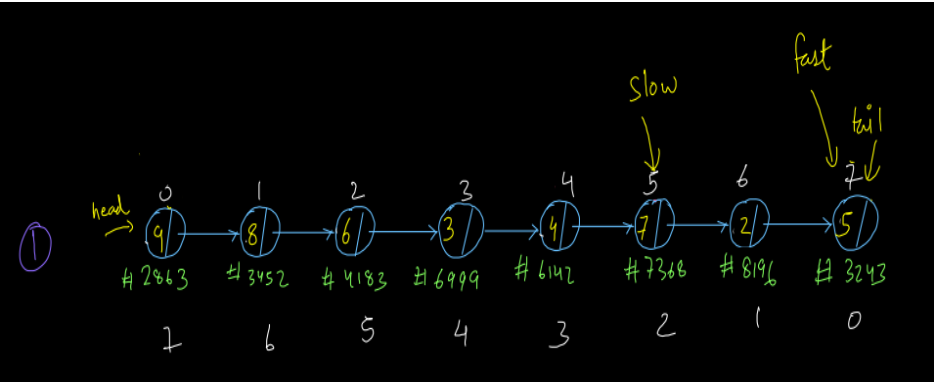
● This is the basis for the two pointer approach. Two Pointer Approach on Linked List:

Ok, now let's get back to our question. We have to find the kth element from the end of the list. We will use the above idea and solve this problem.

1. Make two pointers slow and fast and keep them both on the head of the linked list. 2. Now, move the fast pointer k steps away from the slow pointer. (see fig-5)

****

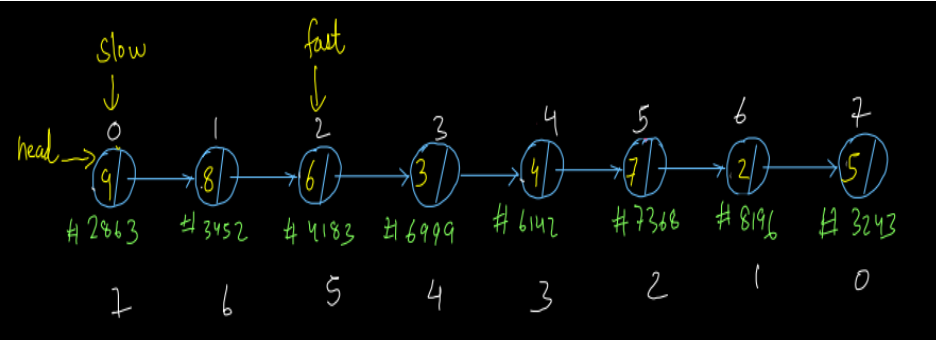
3. Now, move the fast pointer and the slow pointer each one step at a time till the fast pointer reaches the tail of the linked list. When the fast pointer reaches the tail, the node at which the slow pointer resides will be our answer. (See fig-6)

****

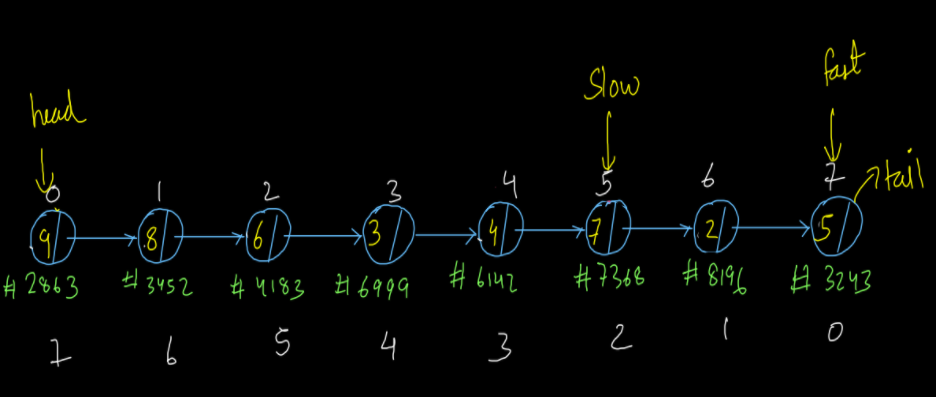
Now that we have understood the entire procedure, let us write the code for the same.

Pseudo Code:

1. Make two pointers slow and fast and keep slow at the head. Initially keep a gap of k between fast and slow pointer i.e. keep the fast pointer "k" steps away from the slow pointer.

****

2. Now, move both slow and fast pointers one step ahead each till fast reaches the end of the list i.e. fast=tail. When fast reaches the last node then wherever slow is at, that will be our "kth" element from the end.

****

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

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

LinkedList list = new LinkedList();

String str = br.readLine();

while (str.equals("quit") == false) {

if (str.startsWith("addLast")) {

int val = Integer.parseInt(str.split(" ")[1]);

list.addLast(val);

} else if (str.startsWith("size")) {

System.out.println(list.size());

} else if (str.startsWith("display")) {

list.display();

} else if (str.startsWith("removeFirst")) {

list.removeFirst();

} else if (str.startsWith("getFirst")) {

int val = list.getFirst();

if (val != -1) {

System.out.println(val);

}

} else if (str.startsWith("getLast")) {

int val = list.getLast();

if (val != -1) {

System.out.println(val);

}

} else if (str.startsWith("getAt")) {

int idx = Integer.parseInt(str.split(" ")[1]);

int val = list.getAt(idx);

if (val != -1) {

System.out.println(val);

}

} else if (str.startsWith("addFirst")) {

int val = Integer.parseInt(str.split(" ")[1]);

list.addFirst(val);

} else if (str.startsWith("addAt")) {

int idx = Integer.parseInt(str.split(" ")[1]);

int val = Integer.parseInt(str.split(" ")[2]);

list.addAt(idx, val);

} else if (str.startsWith("removeLast")) {

list.removeLast();

} else if (str.startsWith("removeAt")) {

int idx = Integer.parseInt(str.split(" ")[1]);

list.removeAt(idx);

} else if (str.startsWith("reverseDI")) {

list.reverseDI();

} else if (str.startsWith("reversePI")) {

list.reversePI();

} else if (str.startsWith("kthFromEnd")) {

int idx = Integer.parseInt(str.split(" ")[1]);

System.out.println(list.kthFromLast(idx));

}

str = br.readLine();

}

}

}

For more clarity of the question, watch the question video

Play Video

Don't get scared by seeing this code. You just have to write one function i.e. kthFromLast(int k) in this. So, hope that you have understood the entire procedure and the code. You may refer to the complete solution video if you have any doubts regarding anything explained above. Hope you also enjoyed the idea behind the two pointer approach and could relate programming to a more real world scenario. Now let us go to the time and space complexity analysis.

**4. Analysis:**

Time Complexity:

The time complexity of the above solution is O(n) as we have traversed the linked list once. We have traversed it in two parts. First we traversed k elements and then we traversed the remaining (size minus k) elements.

Space Complexity:

The space complexity is O(1) as we have not used any extra space for the solution. So, dear reader, with this we have completed this problem. Hope you are clear with everything. If you still have any doubts, you may refer to the complete solution video to clear them.

**5. Suggestions:**

Here are some suggestions from our side that you don't want to miss: The two pointer approach will be used in a lot of problems in Linked List, here and in Level 2 of our course also. So, it is very important to understand the basic idea behind this approach as it is very easy and relates programming even more to the real world. The question wanted us to study the two pointer approach and hence the constraints of not using the size property were provided. You can also solve this problem by using the size property in the same time complexity. We suggest you try this on your own. Hint: Try to relate the index of each node with its index value in the reverse direction. In fig-1 the indices of the nodes going forward and backward, both are shown. Try to take hints from that. Till then, Happy Coding!!!!