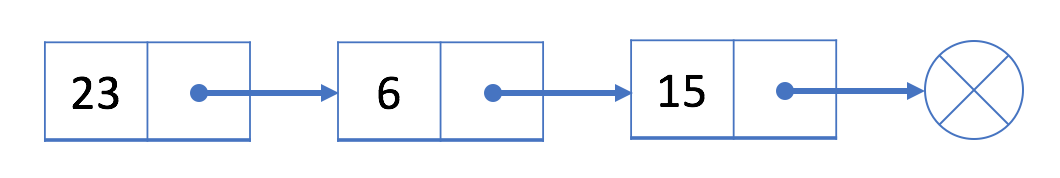
Linked List

**Introduction**

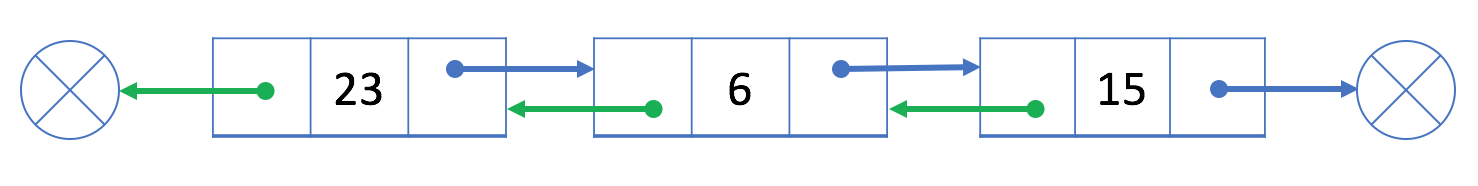
In this card, we are going to introduce another data structure - Linked List.

Similar to the array, the linked list is also a linear data structure. Here is an example:



As you can see, each element in the linked list is actually a separate object while all the objects are linked together by the reference field in each element.

There are two types of linked list: singly linked list and doubly linked list. The example above is a singly linked list and here is an example of doubly linked list:



We will introduce more in later chapters. After this card, you will:

* Understand the structure of singly linked list and doubly linked list;
* Implement traversal, insertion, deletion in a singly or doubly linked list;
* Analyze the complexity of different operations in a singly or doubly linked list;
* Use two-pointer technique (fast-pointer-slow-pointer technique) in the linked list;
* Solve classic problems such as reverse a linked list;
* Analyze the complexity of the algorithms you designed;
* Accumulate experience in designing and debugging.

Singly Linked List

 Introduction - Singly Linked List

 Add Operation - Singly Linked List

 Delete Operation - Singly Linked List

 Design Linked List

 Design Singly Linked List - Solution

Two Pointer Technique

 Two-Pointer in Linked List

 Linked List Cycle

 Linked List Cycle II

 Intersection of Two Linked Lists

 Remove Nth Node From End of List

 Summary - Two-Pointer in Linked List

Classic Problems

 Reverse Linked List

 Reverse Linked List

 Reverse Linked List - Solution

 Remove Linked List Elements

 Odd Even Linked List

 Palindrome Linked List

 Summary - Linked List Classic Problems

Doubly Linked List

 Introduction - Doubly Linked List

 Add Operation - Doubly Linked List

 Delete Operation - Doubly Linked List

 Design Linked List

 Design Doubly Linked List - Solution

Conclusion

 Summary - Linked List

 Merge Two Sorted Lists

 Add Two Numbers

 Flatten a Multilevel Doubly Linked List

 Insert into a Cyclic Sorted List

 Copy List with Random Pointer

 Rotate List

**Singly Linked List**

As we mentioned in the overview, linked list is a linear data structure which link all the separated elements together by the reference field. There are two commonly-used linked list: singly-linked list and doubly-linked list.

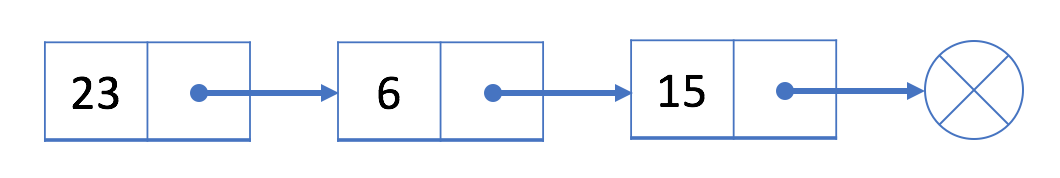
In this chapter, we will start with the singly-linked list and help you:

* Understand the structure of the singly-linked list;
* Perform traversal, insertion and deletion in a singly-linked list;
* Analyze the time complexity of different operations in the singly-linked list;

**Introduction - Singly Linked List**

Each node in a singly-linked list contains not only the value but also a reference field to link to the next node. By this way, the singly-linked list organizes all the nodes in a sequence.

Here is an example of a singly-linked list:



The blue arrows show how nodes in a singly linked list are combined together.

### ***Node Structure***

Here is the typical definition of a node in a singly-linked list:

|  |
| --- |
| // Definition for singly-linked list.  public class SinglyListNode {  int val;  SinglyListNode next;  SinglyListNode(int x) { val = x; }  } |

In most cases, we will use the head node (the first node) to represent the whole list.

### ***Operations***

Unlike the array, we are not able to access a random element in a singly-linked list in constant time. If we want to get the ith element, we have to traverse from the head node one by one. It takes us O(N) time on average to visit an element by index, where N is the length of the linked list.

For instance, in the example above, the head is the node 23. The only way to visit the 3rd node is to use the "next" field of the head node to get to the 2nd node (node 6); Then with the "next" field of node 6, we are able to visit the 3rd node.

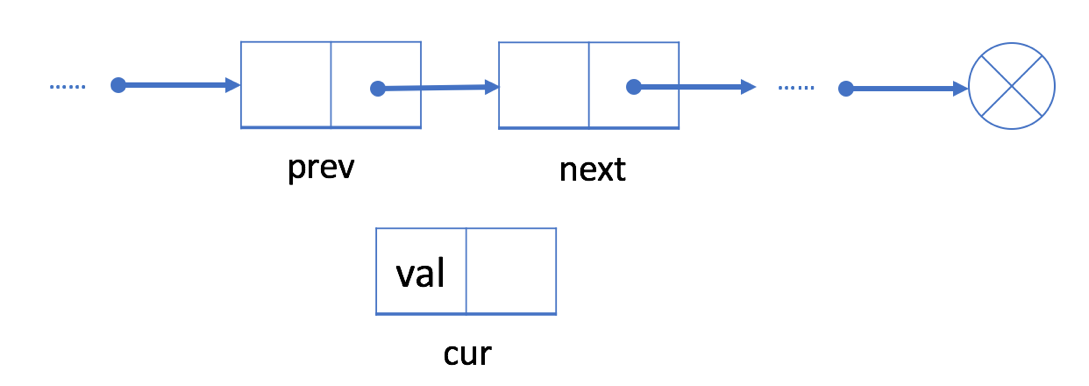
You might wonder why the linked list is useful though it has such a bad performance (compared to the array) in accessing data by index. We will introduce the insert and delete operations in next two articles and you will realize the benefit of the linked list.

After that, we provide an exercise for you to design your own singly linked list.

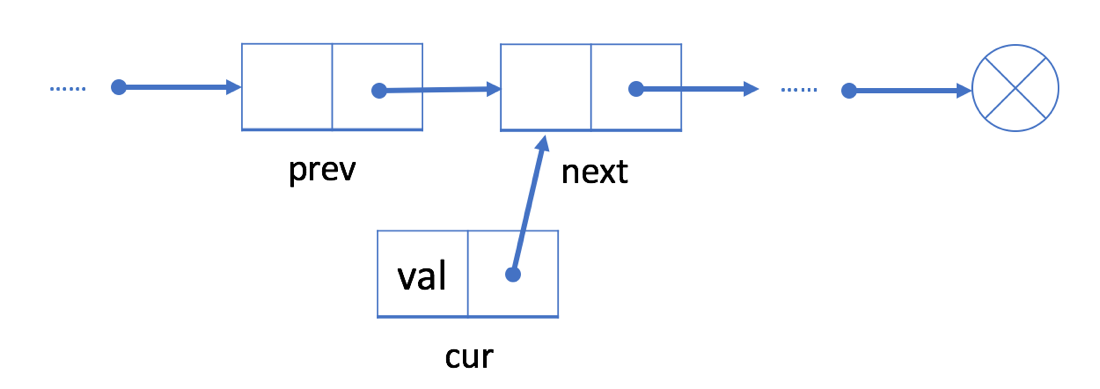
**Add Operation - Singly Linked List**

If we want to add a new value after a given node prev, we should:

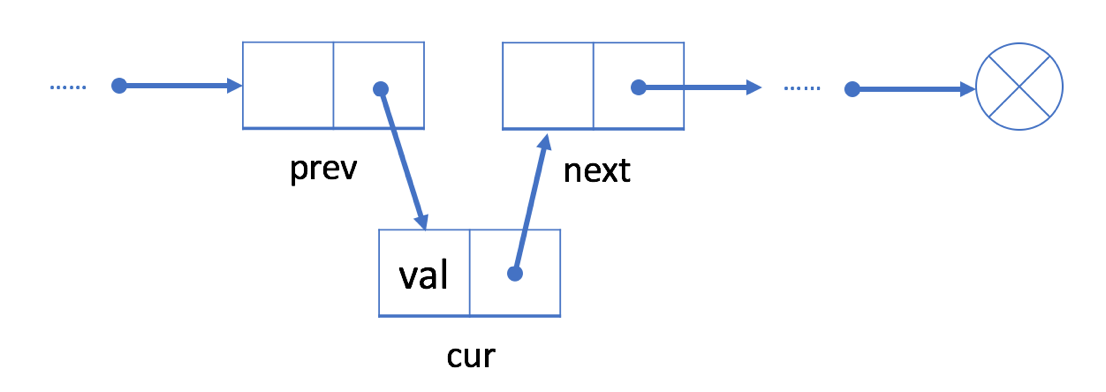
1 Initialize a new node cur with the given value;



2 Link the "next" field of cur to prev's next node next

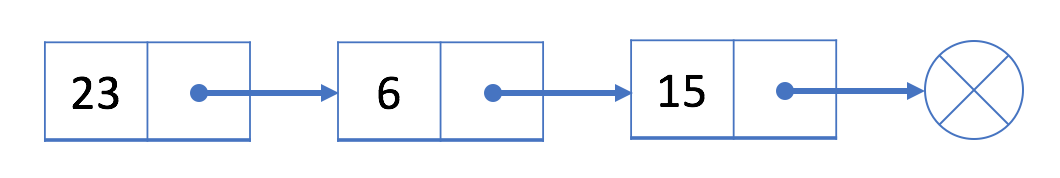


3 Link the "next" field in prev to cur



Unlike an array, we don’t need to move all elements past the inserted element. Therefore, you can insert a new node into a linked list in O(1) time complexity, which is very efficient.

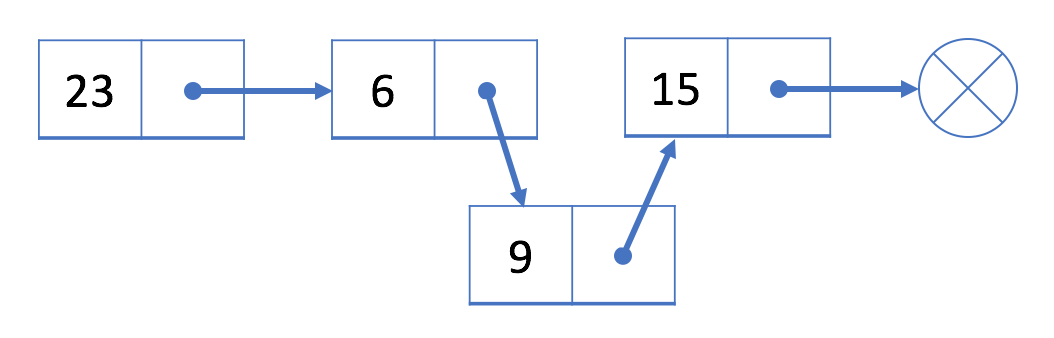
### ***An Example***



Let's insert a new value 9 after the second node 6.

We will first initialize a new node with value 9. Then link node 9 to node 15. Finally, link node 6 to node 9.

After insertion, our linked list will look like this:



### 

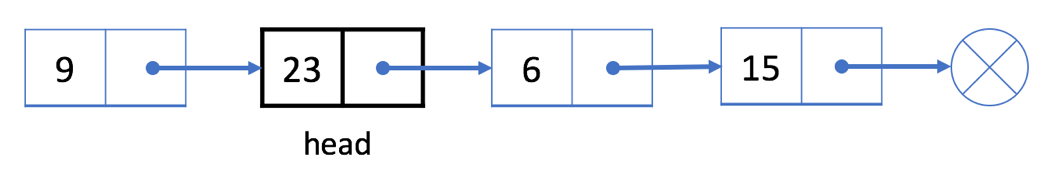
### ***Add a Node at the Beginning***

As we know, we use the head node head to represent the whole list.

So it is essential to update head when adding a new node at the beginning of the list.

1. Initialize a new node cur;
2. Link the new node to our original head node head.
3. Assign cur to head.

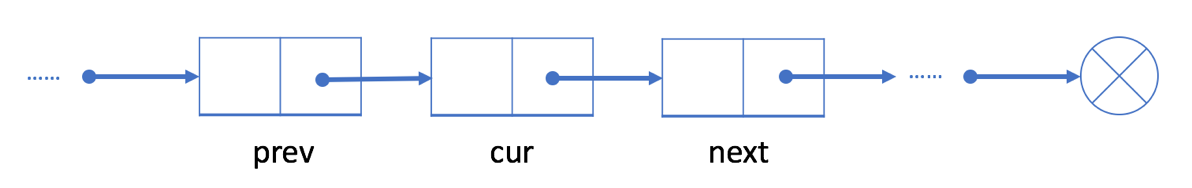
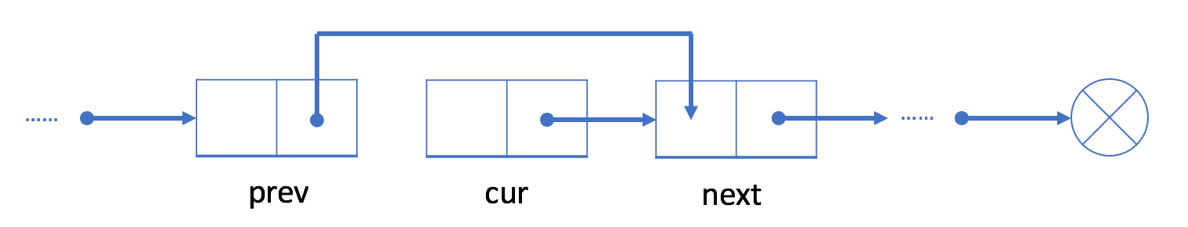
For example, let's add a new node 9 at the beginning of the list.

1. We initialize a new node 9 and link node 9 to current head node 23 
2. Assign node 9 to be our new head. 

What about adding a new node at the end of the list? Can we still use similar strategy?

**Delete Operation - Singly Linked List**

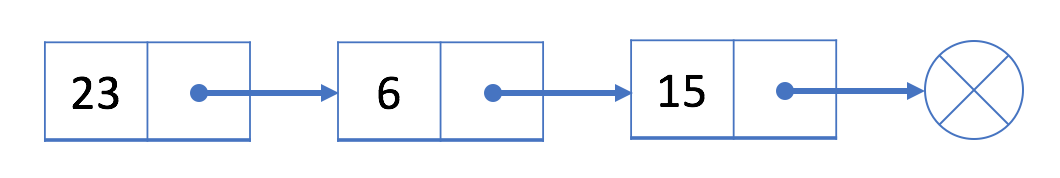
If we want to delete an existing node cur from the singly linked list, we can do it in two steps:

1. Find cur's previous node prev and its next node next. 
2. Link prev to cur's next node next. 

In our first step, we need to find out prev and next. It is easy to find out next using the reference field of cur. However, we have to traverse the linked list from the head node to find out prev which will take O(N) time on average, where N is the length of the linked list. So the time complexity of deleting a node will be O(N).

The space complexity is O(1) because we only need constant space to store our pointers.

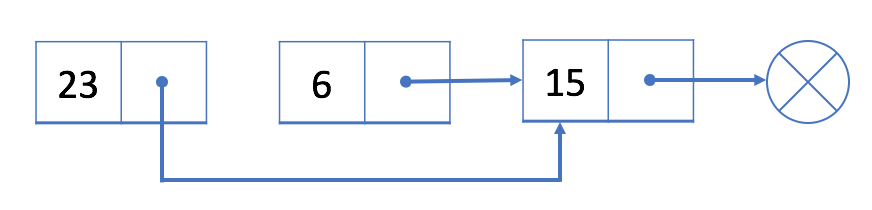
### ***An Example***



Let's try to delete node 6 from the singly linked list above.

1. Traverse the linked list from the head until we find the previous node prev which is node 23

2. Link prev (node 23) with next (node 15)

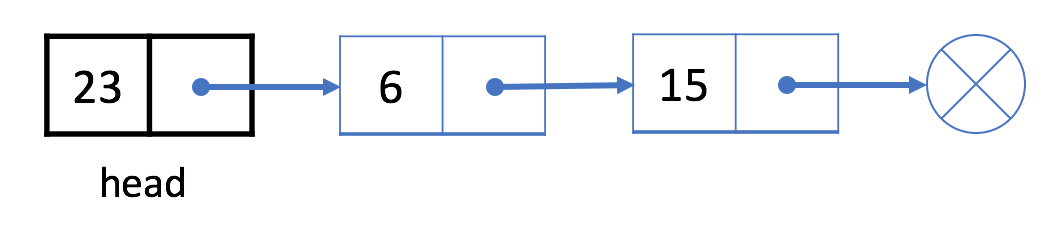


Node 6 is not in our singly linked list now.

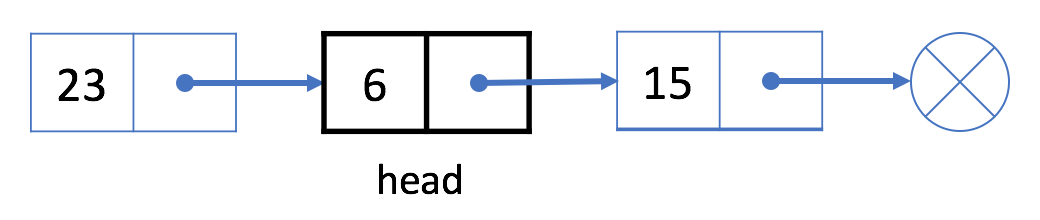
### ***Delete the First Node***

If we want to delete the first node, the strategy will be a little different.

As we mentioned before, we use the head node head to represent a linked list. Our head is the black node 23 in the example below.



If we want to delete the first node, we can simply assign the next node to head. That is to say, our head will be node 6 after deletion.



The linked list begins at the head node, so node 23 is no longer in our linked list.

What about deleting the last node? Can we still use similar strategy?

**Design Linked List**

Design your implementation of the linked list. You can choose to use a singly or doubly linked list.  
A node in a singly linked list should have two attributes: val and next. val is the value of the current node, and next is a pointer/reference to the next node.  
If you want to use the doubly linked list, you will need one more attribute prev to indicate the previous node in the linked list. Assume all nodes in the linked list are **0-indexed**.

Implement the MyLinkedList class:

* MyLinkedList() Initializes the MyLinkedList object.
* int get(int index) Get the value of the indexth node in the linked list. If the index is invalid, return -1.
* void addAtHead(int val) Add a node of value val before the first element of the linked list. After the insertion, the new node will be the first node of the linked list.
* void addAtTail(int val) Append a node of value val as the last element of the linked list.
* void addAtIndex(int index, int val) Add a node of value val before the indexth node in the linked list. If index equals the length of the linked list, the node will be appended to the end of the linked list. If index is greater than the length, the node **will not be inserted**.
* void deleteAtIndex(int index) Delete the indexth node in the linked list, if the index is valid.

**Input**

["MyLinkedList", "addAtHead", "addAtTail", "addAtIndex", "get", "deleteAtIndex", "get"]

[[], [1], [3], [1, 2], [1], [1], [1]]

**Output**

[null, null, null, null, 2, null, 3]

**Explanation**

MyLinkedList myLinkedList = new MyLinkedList();

myLinkedList.addAtHead(1);

myLinkedList.addAtTail(3);

myLinkedList.addAtIndex(1, 2); // linked list becomes 1->2->3

myLinkedList.get(1); // return 2

myLinkedList.deleteAtIndex(1); // now the linked list is 1->3

myLinkedList.get(1); // return 3

**Constraints:**

* 0 <= index, val <= 1000
* Please do not use the built-in LinkedList library.
* At most 2000 calls will be made to get, addAtHead, addAtTail,  addAtIndex and deleteAtIndex.

## Solution

#### **Interview Strategy**

[Linked List](https://en.wikipedia.org/wiki/Linked_list#Basic_concepts_and_nomenclature) is a data structure with zero or several elements. Each element contains a value and link(s) to the other element(s). Depending on the number of links, that could be singly linked list, doubly linked list and multiply linked list.

Singly linked list is the simplest one, it provides addAtHead in a constant time, and addAtTail in a linear time. Though doubly linked list is the most used one, because it provides both addAtHead and addAtTail in a constant time, and optimises the insert and delete operations.

Doubly linked list is implemented in Java as [LinkedList](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedList.html). Since these structures are quite well-known, a good interview strategy would be to mention them during the discussion but not to base the code on them. Better to use the limited interview time to work with two ideas:

* [Sentinel nodes](https://leetcode.com/articles/remove-linked-list-elements/)

Sentinel nodes are widely used in the trees and linked lists as pseudo-heads, pseudo-tails, etc. They serve as the guardians, as the name suggests, and usually they do not hold any data.

Sentinels nodes will be used here to simplify insert and delete. We would apply this in both of the following approaches.

* Bidirectional search for doubly-linked list

Rather than starting from the head, we could search the node in a doubly-linked list from both head and tail.

If you are familiar with the concepts, you can start directly from the Approach #2. By the way, the Approach #2 is 90% of what you need to solve the problem of [LRU Cache](https://leetcode.com/articles/lru-cache/).

#### **Approach 1: Singly Linked List**

Let's start from the simplest possible MyLinkedList, which contains just a structure size and a sentinel head.



|  |
| --- |
| class MyLinkedList {  int size;  ListNode head; // sentinel node as pseudo-head  public MyLinkedList() {  size = 0;  head = new ListNode(0);  }  } |

Note, that sentinel node is used as a pseudo-head and is always present. This way the structure could never be empty, it will contain at least a sentinel head. All nodes in MyLinkedList have a type ListNode: value + link to the next element.

|  |
| --- |
| public class ListNode {  int val;  ListNode next;  ListNode(int x) { val = x; }  } |

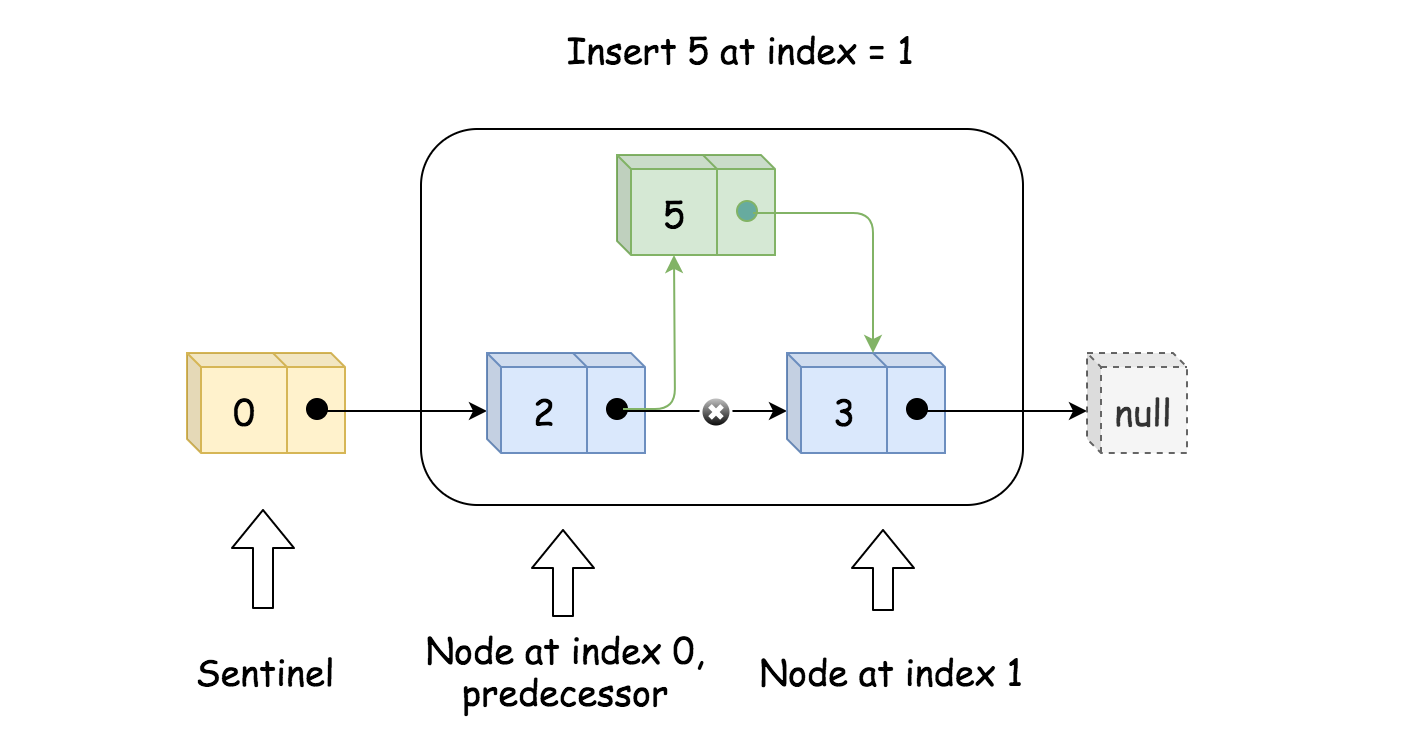
**Add at Index, Add at Head and Add at Tail**

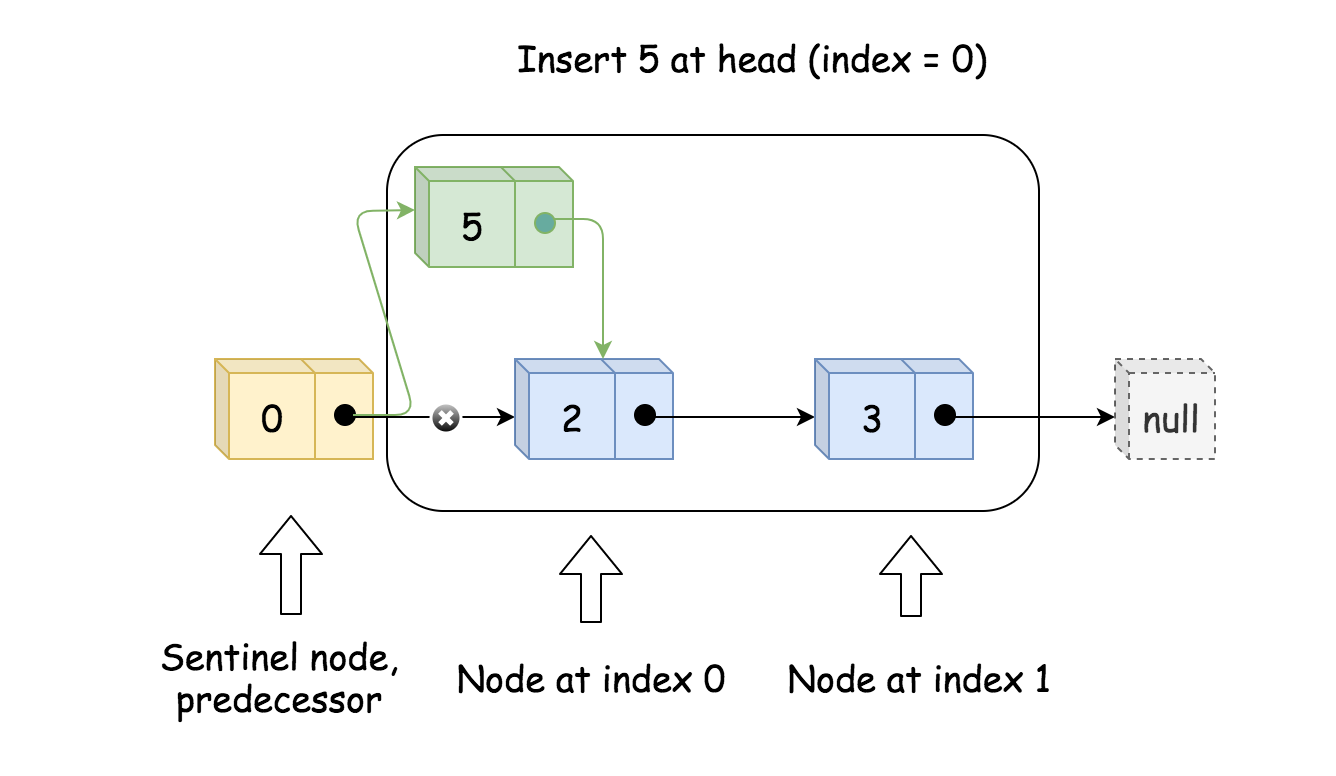
Let's first discuss insert at index operation, because thanks to the sentinel node addAtTail and addAtHead operations could be reduced to this operation as well.

The idea is straightforward:

* Find the predecessor of the node to insert. If the node is to be inserted at head, its predecessor is a sentinel head. If the node is to be inserted at tail, its predecessor is the last node.
* Insert the node by changing the link to the next node.

|  |
| --- |
| toAdd.next = pred.next;  pred.next = toAdd; |



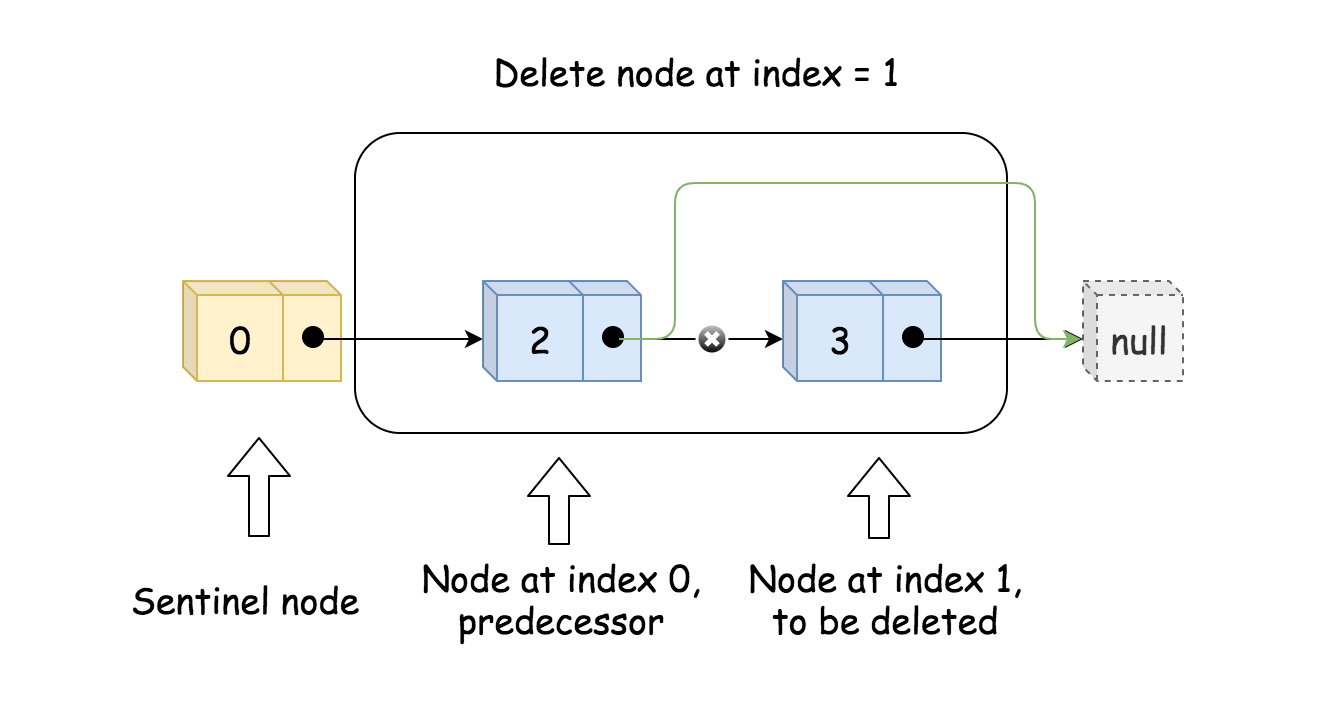


**Delete at Index**

Basically, the same as insert:

* Find the predecessor.
* Delete node by changing the links to the next node.

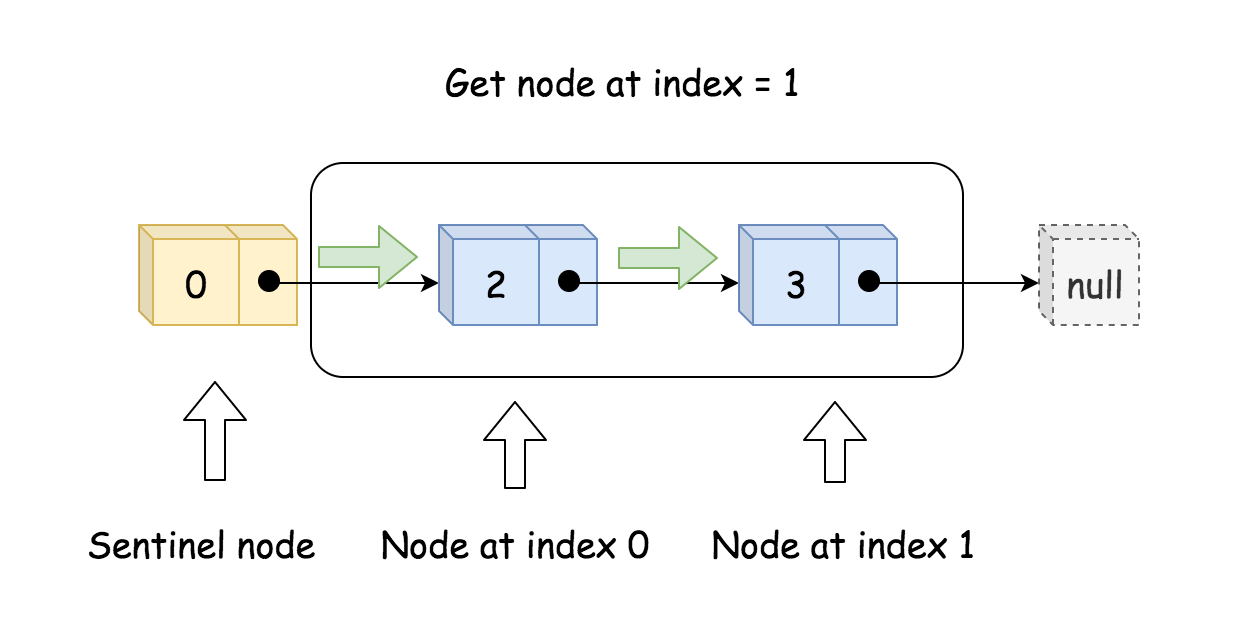
|  |
| --- |
| // delete pred.next  pred.next = pred.next.next; |



**Get**

Get is a very straightforward: start from the sentinel node and do index + 1 steps

|  |
| --- |
| // index steps needed  // to move from sentinel node to wanted index  for(int i = 0; i < index + 1; ++i) curr = curr.next;  return curr.val; |



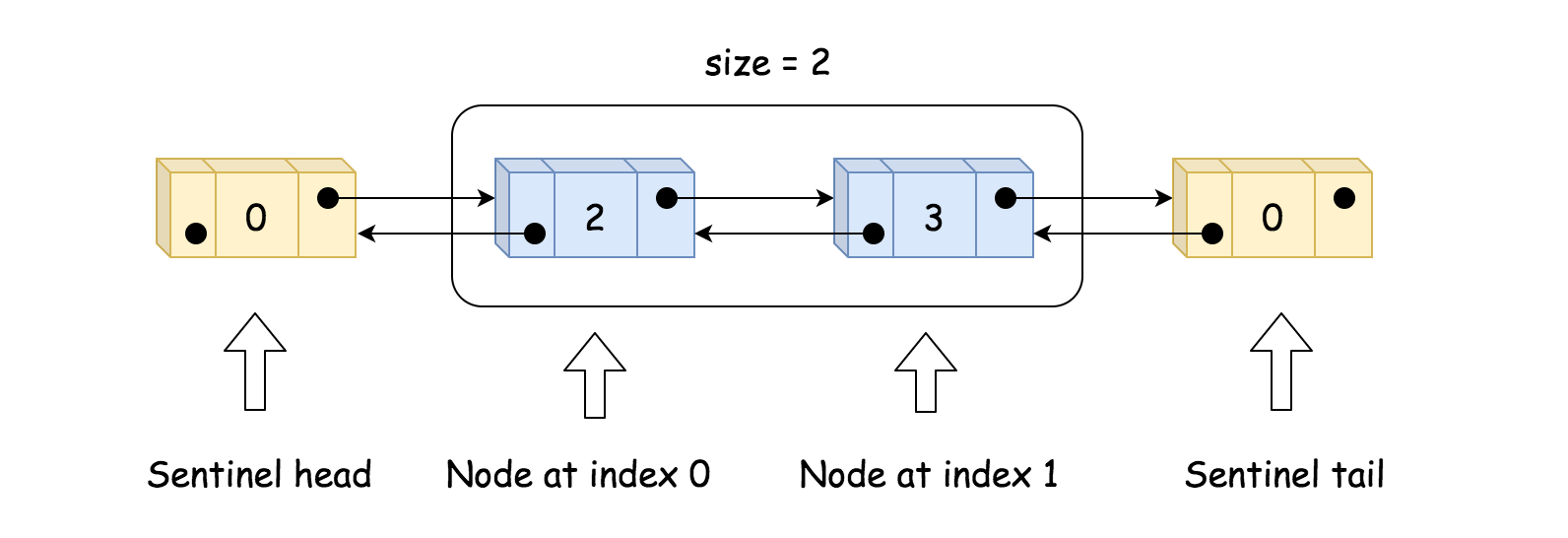
|  |
| --- |
| public class ListNode {  int val;  ListNode next;  ListNode(int x) { val = x; }  }  class MyLinkedList {  int size;  ListNode head; // sentinel node as pseudo-head  public MyLinkedList() {  size = 0;  head = new ListNode(0);  }  /\*\* Get the value of the index-th node in the linked list. If the index is invalid, return -1. \*/  public int get(int index) {  // if index is invalid  if (index < 0 || index >= size) return -1;  ListNode curr = head;  // index steps needed  // to move from sentinel node to wanted index  for(int i = 0; i < index + 1; ++i) curr = curr.next;  return curr.val;  }  /\*\* Add a node of value val before the first element of the linked list. After the insertion, the new node will be the first node of the linked list. \*/  public void addAtHead(int val) {  addAtIndex(0, val);  }  /\*\* Append a node of value val to the last element of the linked list. \*/  public void addAtTail(int val) {  addAtIndex(size, val);  }  /\*\* Add a node of value val before the index-th node in the linked list. If index equals to the length of linked list, the node will be appended to the end of linked list. If index is greater than the length, the node will not be inserted. \*/  public void addAtIndex(int index, int val) {  // If index is greater than the length,  // the node will not be inserted.  if (index > size) return;  // [so weird] If index is negative,  // the node will be inserted at the head of the list.  if (index < 0) index = 0;  ++size;  // find predecessor of the node to be added  ListNode pred = head;  for(int i = 0; i < index; ++i) pred = pred.next;  // node to be added  ListNode toAdd = new ListNode(val);  // insertion itself  toAdd.next = pred.next;  pred.next = toAdd;  }  /\*\* Delete the index-th node in the linked list, if the index is valid. \*/  public void deleteAtIndex(int index) {  // if the index is invalid, do nothing  if (index < 0 || index >= size) return;  size--;  // find predecessor of the node to be deleted  ListNode pred = head;  for(int i = 0; i < index; ++i) pred = pred.next;  // delete pred.next  pred.next = pred.next.next;  }  } |

**Complexity Analysis**

* Time complexity: O(1) for addAtHead. O(*k*) for get, addAtIndex, and deleteAtIndex, where k*k* is an index of the element to get, add or delete. O(*N*) for addAtTail.
* Space complexity: O(1) for all operations.

#### **Approach 2: Doubly Linked List**

Time to implement DLL MyLinkedList, which is a much faster (twice faster on the testcase set here) though a bit more complex. It contains size, sentinel head and sentinel tail.



|  |
| --- |
| class MyLinkedList {  int size;  // sentinel nodes as pseudo-head and pseudo-tail  ListNode head, tail;  public MyLinkedList() {  size = 0;  head = new ListNode(0);  tail = new ListNode(0);  head.next = tail;  tail.prev = head;  }  } |

Note, that sentinel head and tail are always present. All nodes in MyLinkedList have a type ListNode: value + two links: to the next and to the previous elements.

|  |
| --- |
| public class ListNode {  int val;  ListNode next;  ListNode prev;  ListNode(int x) { val = x; }  } |

**Add at Index, Add at Head and Add at Tail**

The idea is simple:

* Find the predecessor and the successor of the node to insert. If the node is to be inserted at head, its predecessor is a sentinel head. If the node is to be inserted at tail, its successor is a sentinel tail.

Use bidirectional search to perform faster.

* Insert the node by changing the links to the next and previous nodes.

|  |
| --- |
| * toAdd.prev = pred * toAdd.next = succ * pred.next = toAdd * succ.prev = toAdd |

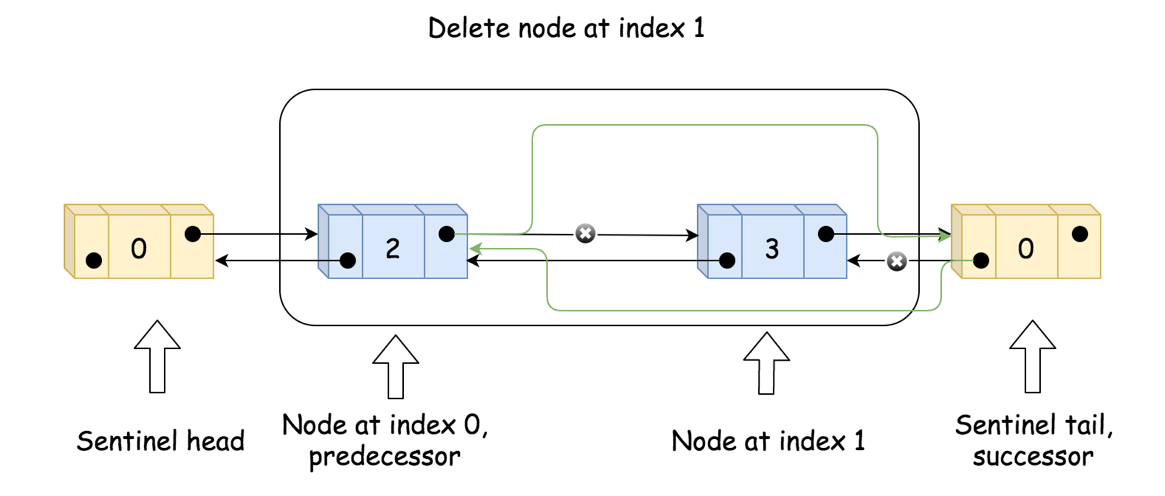


**Delete at Index**

Basically, the same as insert:

* Find the predecessor and successor.
* Delete node by changing the links to the next and previous nodes.

|  |
| --- |
| pred.next = succ  succ.prev = pred |

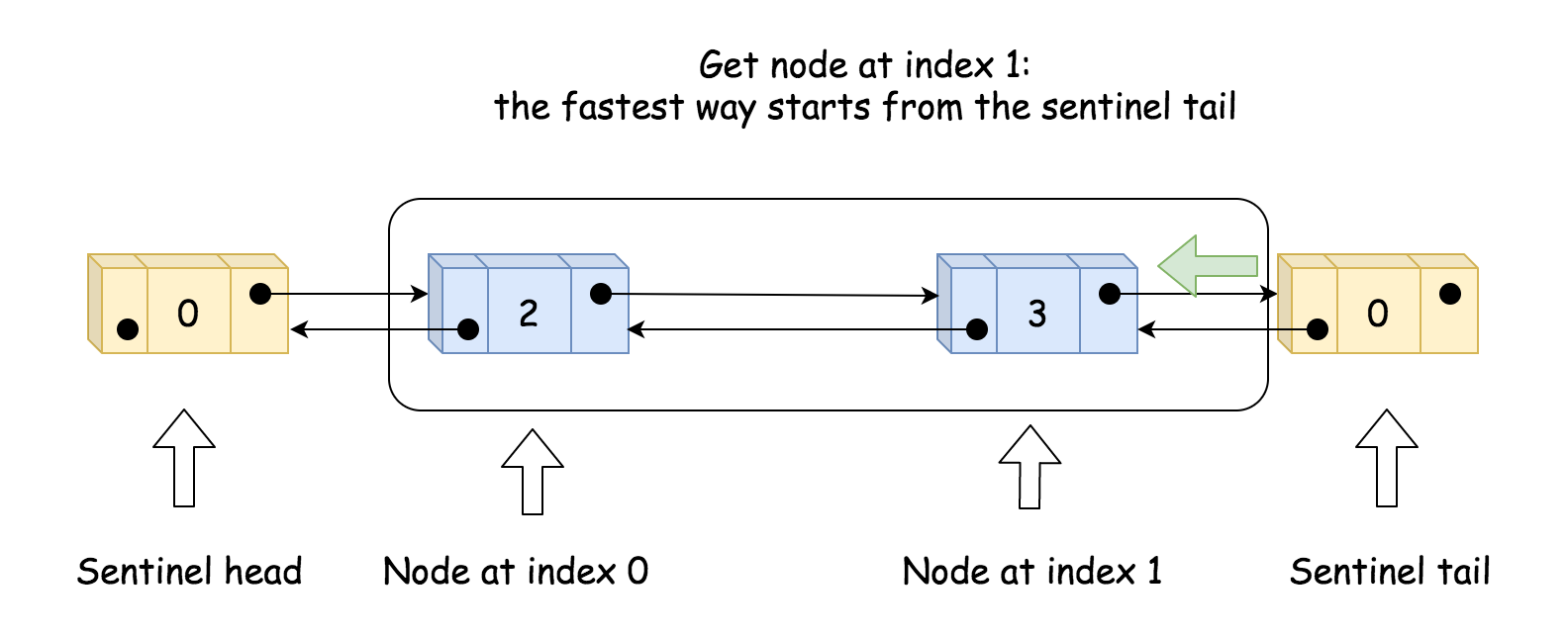


**Get**

Get is very straightforward:

* Compare index and size - index to define the fastest way: starting from the head, or starting from the tail.
* Go to the wanted node.

|  |
| --- |
| // choose the fastest way: to move from the head  // or to move from the tail  ListNode curr = head;  if (index + 1 < size - index)  for(int i = 0; i < index + 1; ++i) curr = curr.next;  else {  curr = tail;  for(int i = 0; i < size - index; ++i) curr = curr.prev;  } |



|  |
| --- |
| public class ListNode {  int val;  ListNode next;  ListNode prev;  ListNode(int x) { val = x; }  }  class MyLinkedList {  int size;  // sentinel nodes as pseudo-head and pseudo-tail  ListNode head, tail;  public MyLinkedList() {  size = 0;  head = new ListNode(0);  tail = new ListNode(0);  head.next = tail;  tail.prev = head;  }  /\*\* Get the value of the index-th node in the linked list. If the index is invalid, return -1. \*/  public int get(int index) {  // if index is invalid  if (index < 0 || index >= size) return -1;  // choose the fastest way: to move from the head  // or to move from the tail  ListNode curr = head;  if (index + 1 < size - index)  for(int i = 0; i < index + 1; ++i) curr = curr.next;  else {  curr = tail;  for(int i = 0; i < size - index; ++i) curr = curr.prev;  }  return curr.val;  }  /\*\* Add a node of value val before the first element of the linked list. After the insertion, the new node will be the first node of the linked list. \*/  public void addAtHead(int val) {  ListNode pred = head, succ = head.next;  ++size;  ListNode toAdd = new ListNode(val);  toAdd.prev = pred;  toAdd.next = succ;  pred.next = toAdd;  succ.prev = toAdd;  }  /\*\* Append a node of value val to the last element of the linked list. \*/  public void addAtTail(int val) {  ListNode succ = tail, pred = tail.prev;  ++size;  ListNode toAdd = new ListNode(val);  toAdd.prev = pred;  toAdd.next = succ;  pred.next = toAdd;  succ.prev = toAdd;  }  /\*\* Add a node of value val before the index-th node in the linked list. If index equals to the length of linked list, the node will be appended to the end of linked list. If index is greater than the length, the node will not be inserted. \*/  public void addAtIndex(int index, int val) {  // If index is greater than the length,  // the node will not be inserted.  if (index > size) return;  // [so weird] If index is negative,  // the node will be inserted at the head of the list.  if (index < 0) index = 0;  // find predecessor and successor of the node to be added  ListNode pred, succ;  if (index < size - index) {  pred = head;  for(int i = 0; i < index; ++i) pred = pred.next;  succ = pred.next;  }  else {  succ = tail;  for (int i = 0; i < size - index; ++i) succ = succ.prev;  pred = succ.prev;  }  // insertion itself  ++size;  ListNode toAdd = new ListNode(val);  toAdd.prev = pred;  toAdd.next = succ;  pred.next = toAdd;  succ.prev = toAdd;  }  /\*\* Delete the index-th node in the linked list, if the index is valid. \*/  public void deleteAtIndex(int index) {  // if the index is invalid, do nothing  if (index < 0 || index >= size) return;  // find predecessor and successor of the node to be deleted  ListNode pred, succ;  if (index < size - index) {  pred = head;  for(int i = 0; i < index; ++i) pred = pred.next;  succ = pred.next.next;  }  else {  succ = tail;  for (int i = 0; i < size - index - 1; ++i) succ = succ.prev;  pred = succ.prev.prev;  }  // delete pred.next  --size;  pred.next = succ;  succ.prev = pred;  }  } |

**Complexity Analysis**

* Time complexity: O(1) for addAtHead and addAtTail. O(min(*k*,*N*−*k*)) for get, addAtIndex, and deleteAtIndex, where *k* is an index of the element to get, add or delete.
* Space complexity: O(1) for all operations.

**Design Singly Linked List - Solution**

Let's briefly review the structure definition of a node in the singly linked list.

|  |
| --- |
| // Definition for singly-linked list.  public class SinglyListNode {  int val;  SinglyListNode next;  SinglyListNode(int x) { val = x; }  } |

Based on this definition, we are going to give you the solution step by step.

**1. Initiate a new linked list: represent a linked list using the head node.**

|  |
| --- |
| class MyLinkedList {  private SinglyListNode head;  /\*\* Initialize your data structure here. \*/  public MyLinkedList() {  head = null;  }  } |

**2. Traverse the linked list to get element by index.**

|  |
| --- |
| /\*\* Helper function to return the index-th node in the linked list. \*/  private SinglyListNode getNode(int index) {  SinglyListNode cur = head;  for (int i = 0; i < index && cur != null; ++i) {  cur = cur.next;  }  return cur;  }  /\*\* Helper function to return the last node in the linked list. \*/  private SinglyListNode getTail() {  SinglyListNode cur = head;  while (cur != null && cur.next != null) {  cur = cur.next;  }  return cur;  }  /\*\* Get the value of the index-th node in the linked list. If the index is invalid, return -1. \*/  public int get(int index) {  SinglyListNode cur = getNode(index);  return cur == null ? -1 : cur.val;  } |

**3. Add a new node.**

|  |
| --- |
| /\*\* Add a node of value val before the first element of the linked list. After the insertion, the new node will be the first node of the linked list. \*/  public void addAtHead(int val) {  SinglyListNode cur = new SinglyListNode(val);  cur.next = head;  head = cur;  return;  }  /\*\* Append a node of value val to the last element of the linked list. \*/  public void addAtTail(int val) {  if (head == null) {  addAtHead(val);  return;  }  SinglyListNode prev = getTail();  SinglyListNode cur = new SinglyListNode(val);  prev.next = cur;  }  /\*\* Add a node of value val before the index-th node in the linked list. If index equals to the length of linked list, the node will be appended to the end of linked list. If index is greater than the length, the node will not be inserted. \*/  public void addAtIndex(int index, int val) {  if (index == 0) {  addAtHead(val);  return;  }  SinglyListNode prev = getNode(index - 1);  if (prev == null) {  return;  }  SinglyListNode cur = new SinglyListNode(val);  SinglyListNode next = prev.next;  cur.next = next;  prev.next = cur;  } |

It is worth noting that we have to get the (index - 1)th node or the last node before we add the new node (except adding at the head) and it takes O(N) time to get a node by index, where *N* is the length of the linked list. It is different from adding a new node after a given node.

**5. Delete a node.**

|  |
| --- |
| /\*\* Delete the index-th node in the linked list, if the index is valid. \*/  public void deleteAtIndex(int index) {  SinglyListNode cur = getNode(index);  if (cur == null) {  return;  }  SinglyListNode prev = getNode(index - 1);  SinglyListNode next = cur.next;  if (prev != null) {  prev.next = next;  } else {  // modify head when deleting the first node.  head = next;  }  } |

Similar to the add operation, it takes O(N) time to get the node by the index which is different from deleting a given node. However, even if we already get the node we want to delete, we still have to traverse to get its previous node.

## Two Pointer Technique

We have introduced the two-pointer technique in another card: [Introduction to Data Structure - Array](https://leetcode.com/explore/learn/card/array-and-string/205/array-two-pointer-technique/).

Let's briefly review this technique. We mentioned two scenarios to use the two-pointer technique:

1. Two pointers starts at different position: one starts at the beginning while another starts at the end;
2. Two pointers are moved at different speed: one is faster while another one might be slower.

For a singly linked list, since we can only traverse the linked list in one direction, the first scenario might not work. However, the second scenario, which is also called slow-pointer and fast-pointer technique, is really useful.

In this chapter, we will focus on slow-pointer and fast-pointer problem in the linked list and show you how to solve this problem.

**Two-Pointer in Linked List**

Let's start with a classic problem:

Given a linked list, determine if it has a cycle in it.

You might have come up with the solution using the hash table. But there is a more efficient solution using the two-pointer technique. Try to think it over by yourself before reading the remaining content.

Imagine there are two runners with different speed. If they are running on a straight path, the fast runner will first arrive at the destination. However, if they are running on a circular track, the fast runner will catch up with the slow runner if they keep running.

That's exactly what we will come across using two pointers with different speed in a linked list:

1. If there is no cycle, the fast pointer will stop at the end of the linked list.
2. If there is a cycle, the fast pointer will eventually meet with the slow pointer.

So the only remaining problem is:

What should be the proper speed for the two pointers?

It is a safe choice to move the slow pointer one step at a time while moving the fast pointer two steps at a time. For each iteration, the fast pointer will move one extra step. If the length of the cycle is *M*, after *M* iterations, the fast pointer will definitely move one more cycle and catch up with the slow pointer.

What about other choices? Do they work? Would they be more efficient?

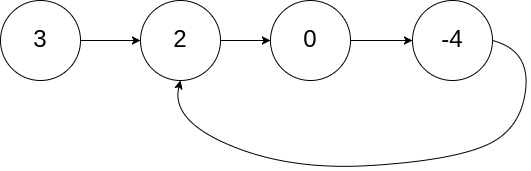
**Linked List Cycle**

Given head, the head of a linked list, determine if the linked list has a cycle in it.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to. **Note that pos is not passed as a parameter**.

Return true*if there is a cycle in the linked list*. Otherwise, return false.

**Example 1:**

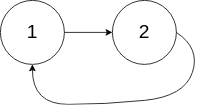


**Input:** head = [3,2,0,-4], pos = 1

**Output:** true

**Explanation:** There is a cycle in the linked list, where the tail connects to the 1st node (0-indexed).

**Example 2:**



**Input:** head = [1,2], pos = 0

**Output:** true

**Explanation:** There is a cycle in the linked list, where the tail connects to the 0th node.

**Example 3:**



**Input:** head = [1], pos = -1

**Output:** false

**Explanation:** There is no cycle in the linked list.

**Constraints:**

* The number of the nodes in the list is in the range [0, 104].
* -105 <= Node.val <= 105
* pos is -1 or a **valid index** in the linked-list.

**Follow up:** Can you solve it using O(1) (i.e. constant) memory?

## Summary

This article is for beginners. It introduces the following ideas: Linked List, Hash Table and Two Pointers.

## Solution

#### **Approach 1: Hash Table**

**Intuition**

To detect if a list is cyclic, we can check whether a node had been visited before. A natural way is to use a hash table.

**Algorithm**

We go through each node one by one and record each node's reference (or memory address) in a hash table. If the current node is null, we have reached the end of the list and it must not be cyclic. If current node’s reference is in the hash table, then return true.

|  |
| --- |
| public class Solution {  public boolean hasCycle(ListNode head) {  Set<ListNode> nodesSeen = new HashSet<>();  while (head != null) {  if (nodesSeen.contains(head)) {  return true;  }  nodesSeen.add(head);  head = head.next;  }  return false;  }  } |

**Complexity analysis**

Let n*n* be the total number of nodes in the linked list.

* Time complexity : *O*(*n*). We visit each of the *n* elements in the list at most once. Adding a node to the hash table costs only *O*(1) time.
* Space complexity: *O*(*n*). The space depends on the number of elements added to the hash table, which contains at most *n* elements.

#### **Approach 2: Floyd's Cycle Finding Algorithm**

**Intuition**

Imagine two runners running on a track at different speed. What happens when the track is actually a circle?

**Algorithm**

The space complexity can be reduced to O(1)*O*(1) by considering two pointers at **different speed** - a slow pointer and a fast pointer. The slow pointer moves one step at a time while the fast pointer moves two steps at a time.

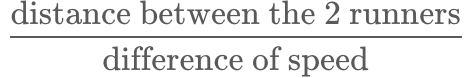
If there is no cycle in the list, the fast pointer will eventually reach the end and we can return false in this case.

Now consider a cyclic list and imagine the slow and fast pointers are two runners racing around a circle track. The fast runner will eventually meet the slow runner. Why? Consider this case (we name it case A) - The fast runner is just one step behind the slow runner. In the next iteration, they both increment one and two steps respectively and meet each other.

How about other cases? For example, we have not considered cases where the fast runner is two or three steps behind the slow runner yet. This is simple, because in the next or next's next iteration, this case will be reduced to case A mentioned above.

|  |
| --- |
| public class Solution {  public boolean hasCycle(ListNode head) {  if (head == null) {  return false;  }  ListNode slow = head;  ListNode fast = head.next;  while (slow != fast) {  if (fast == null || fast.next == null) {  return false;  }  slow = slow.next;  fast = fast.next.next;  }  return true;  }  } |

**Complexity analysis**

* Time complexity : *O*(*n*). Let us denote *n* as the total number of nodes in the linked list. To analyze its time complexity, we consider the following two cases separately.
  + ***List has no cycle:***  
    The fast pointer reaches the end first and the run time depends on the list's length, which is *O*(*n*).
  + ***List has a cycle:***  
    We break down the movement of the slow pointer into two steps, the non-cyclic part and the cyclic part:
    1. The slow pointer takes "non-cyclic length" steps to enter the cycle. At this point, the fast pointer has already reached the cycle. Number of iterations=non-cyclic length=*N*
    2. Both pointers are now in the cycle. Consider two runners running in a cycle - the fast runner moves 2 steps while the slow runner moves 1 steps at a time. Since the speed difference is 1, it takes  loops for the fast runner to catch up with the slow runner. As the distance is at most "cyclic length K" and the speed difference is 1, we conclude that  
       Number of iterations=almost "cyclic length K".

Therefore, the worst case time complexity is *O*(*N*+*K*), which is *O*(*n*).

* Space complexity : *O*(1). We only use two nodes (slow and fast) so the space complexity is *O*(1).

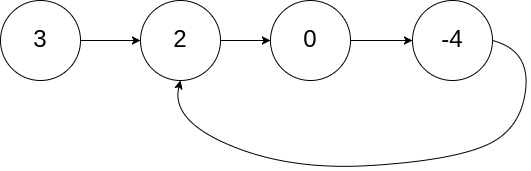
**Linked List Cycle II**

Given a linked list, return the node where the cycle begins. If there is no cycle, return null.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to. **Note that pos is not passed as a parameter**.

**Notice** that you **should not modify** the linked list.

**Example 1:**

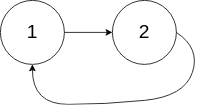


**Input:** head = [3,2,0,-4], pos = 1

**Output:** tail connects to node index 1

**Explanation:** There is a cycle in the linked list, where tail connects to the second node.

**Example 2:**



**Input:** head = [1,2], pos = 0

**Output:** tail connects to node index 0

**Explanation:** There is a cycle in the linked list, where tail connects to the first node.

**Example 3:**



**Input:** head = [1], pos = -1

**Output:** no cycle

**Explanation:** There is no cycle in the linked list.

**Constraints:**

* The number of the nodes in the list is in the range [0, 104].
* -105 <= Node.val <= 105
* pos is -1 or a **valid index** in the linked-list.

**Follow up:** Can you solve it using O(1) (i.e. constant) memory?

#### **Approach 1: Hash Table**

**Intuition**

If we keep track of the nodes that we've seen already in a Set, we can traverse the list and return the first duplicate node.

**Algorithm**

First, we allocate a Set to store ListNode references. Then, we traverse the list, checking visited for containment of the current node. If the node has already been seen, then it is necessarily the entrance to the cycle. If any other node were the entrance to the cycle, then we would have already returned that node instead. Otherwise, the if condition will never be satisfied, and our function will return null.

The algorithm necessarily terminates for any list with a finite number of nodes, as the domain of input lists can be divided into two categories: cyclic and acyclic lists. An acyclic list resembles a null-terminated chain of nodes, while a cyclic list can be thought of as an acyclic list with the final null replaced by a reference to some previous node. If the while loop terminates, we return null, as we have traversed the entire list without encountering a duplicate reference. In this case, the list is acyclic. For a cyclic list, the while loop will never terminate, but at some point the if condition will be satisfied and cause the function to return.

|  |
| --- |
| public class Solution {  public ListNode detectCycle(ListNode head) {  Set<ListNode> visited = new HashSet<ListNode>();  ListNode node = head;  while (node != null) {  if (visited.contains(node)) {  return node;  }  visited.add(node);  node = node.next;  }  return null;  }  } |

**Complexity Analysis**

* Time complexity : *O*(*n*)

For both cyclic and acyclic inputs, the algorithm must visit each node exactly once. This is transparently obvious for acyclic lists because the *n*th node points to null, causing the loop to terminate. For cyclic lists, the if condition will cause the function to return after visiting the *n*th node, as it points to some node that is already in visited. In both cases, the number of nodes visited is exactly *n*, so the runtime is linear in the number of nodes.

* Space complexity : *O*(*n*)

For both cyclic and acyclic inputs, we will need to insert each node into the Set once. The only difference between the two cases is whether we discover that the "last" node points to null or a previously-visited node. Therefore, because the Set will contain *n* distinct nodes, the memory footprint is linear in the number of nodes.

#### **Approach 2: Floyd's Tortoise and Hare**

**Intuition**

What happens when a fast runner (a hare) races a slow runner (a tortoise) on a circular track? At some point, the fast runner will catch up to the slow runner from behind.

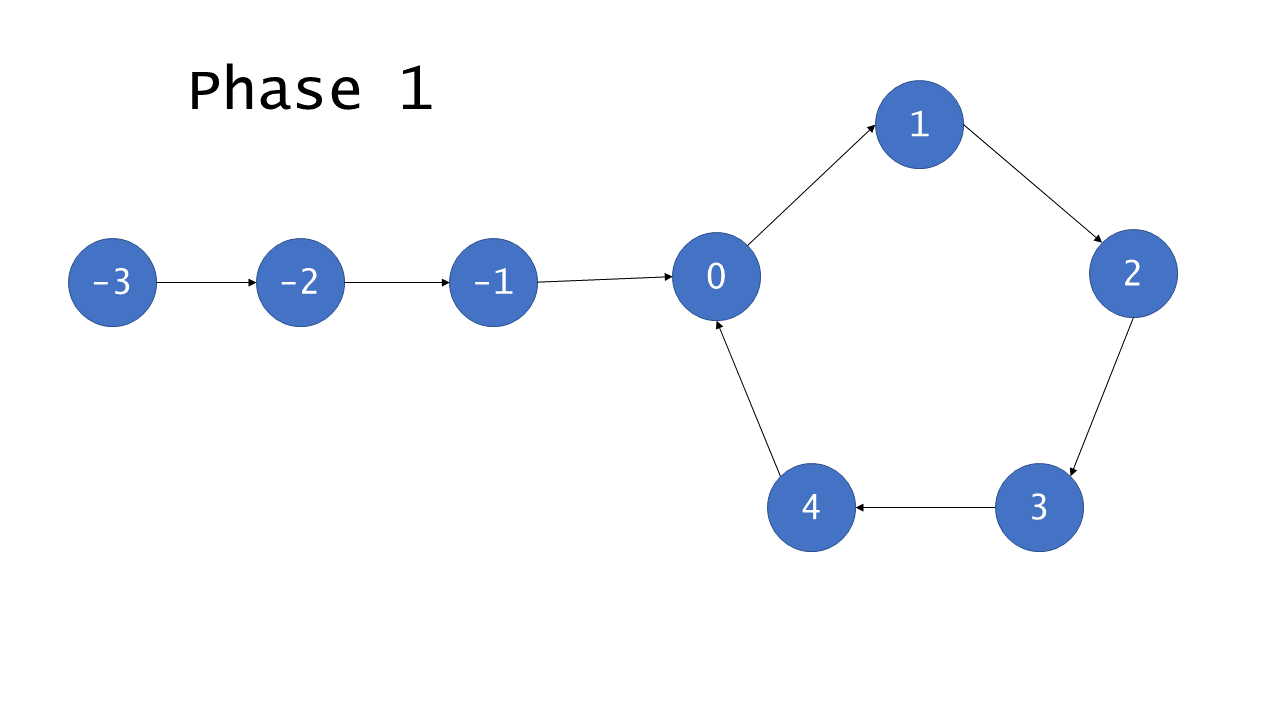
**Algorithm**

Floyd's algorithm is separated into two distinct phases. In the first phase, it determines whether a cycle is present in the list. If no cycle is present, it returns null immediately, as it is impossible to find the entrance to a nonexistant cycle. Otherwise, it uses the located "intersection node" to find the entrance to the cycle.

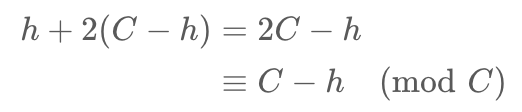
Phase 1

Here, we initialize two pointers - the fast hare and the slow tortoise. Then, until hare can no longer advance, we increment tortoise once and hare twice.[[1]](https://leetcode.com/problems/linked-list-cycle-ii/solution/#fn1) If, after advancing them, hare and tortoise point to the same node, we return it. Otherwise, we continue. If the while loop terminates without returning a node, then the list is acyclic, and we return null to indicate as much.

To see why this works, consider the image below:



Here, the nodes in the cycle have been labelled from 0 to *C*−1, where *C* is the length of the cycle. The noncyclic nodes have been labelled from -−*F* to -1, where *F* is the number of nodes outside of the cycle. After *F* iterations, tortoise points to node 0 and hare points to some node *h*, where C*F* ≡ *h* (mod *C*). This is because hare traverses 2*F* nodes over the course of *F* iterations, exactly *F* of which are in the cycle. After *C*−*h* more iterations, tortoise obviously points to node *C*−*h*, but (less obviously) hare also points to the same node. To see why, remember that hare traverses 2(*C*−*h*) from its starting position of *h*:

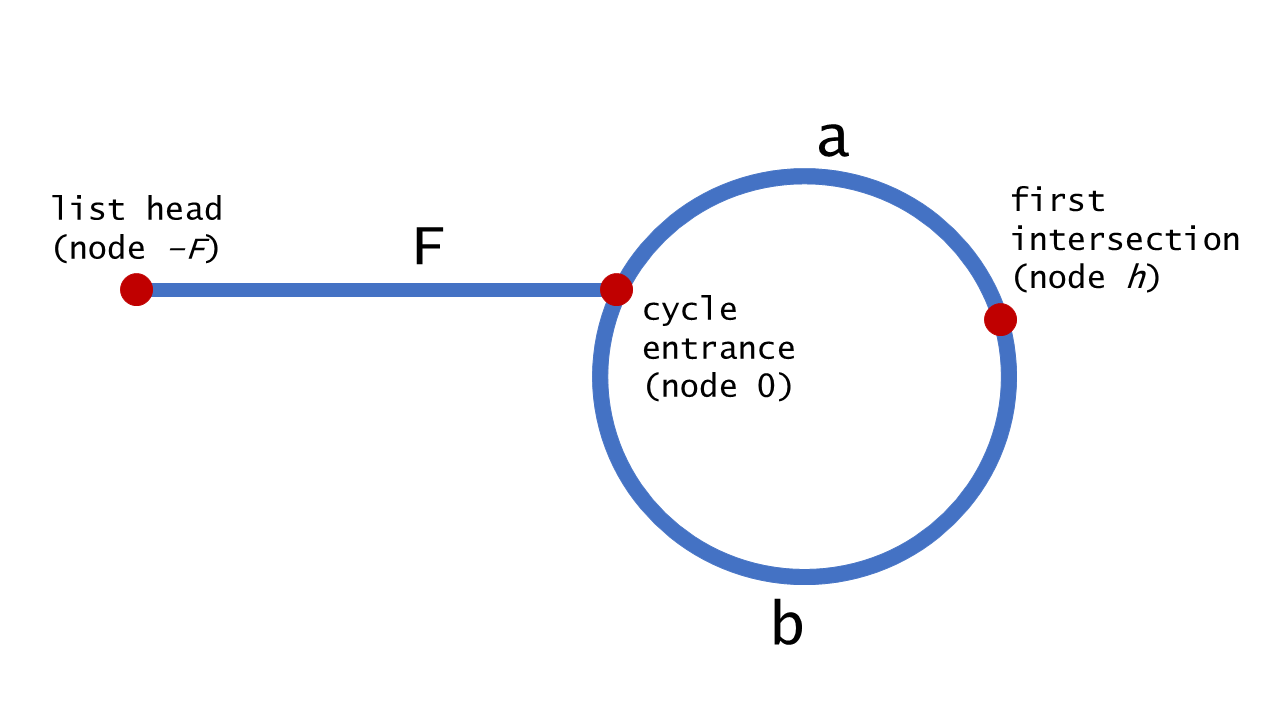


Therefore, given that the list is cyclic, hare and tortoise will eventually both point to the same node, known henceforce as the *intersection*.

*Phase 2*

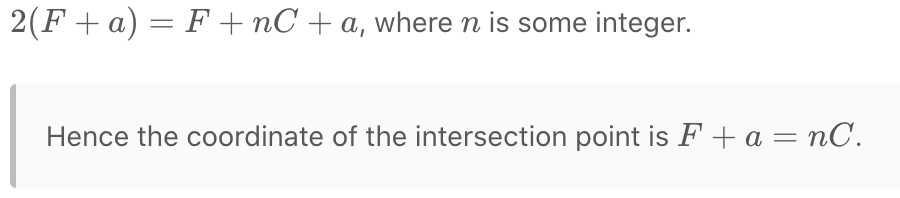
Given that phase 1 finds an intersection, phase 2 proceeds to find the node that is the entrance to the cycle. To do so, we initialize two more pointers: ptr1, which points to the head of the list, and ptr2, which points to the intersection. Then, we advance each of them by 1 until they meet; the node where they meet is the entrance to the cycle, so we return it.

Use the diagram below to help understand the proof of this approach's correctness.

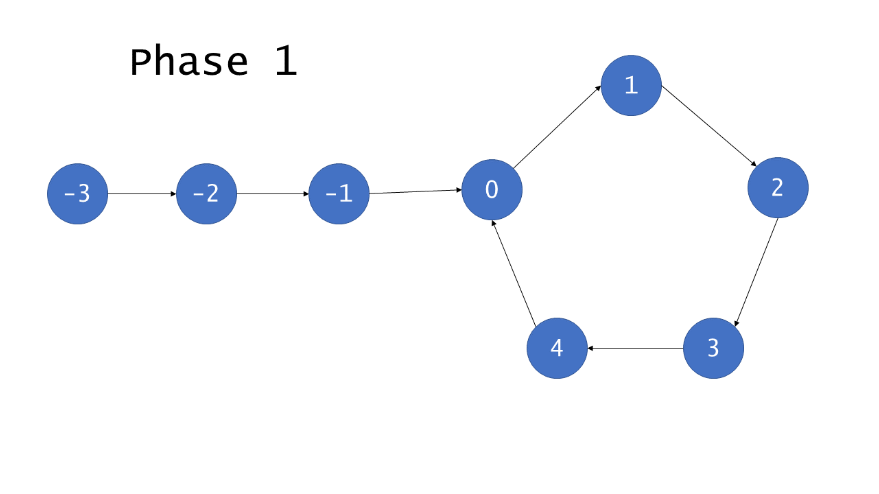


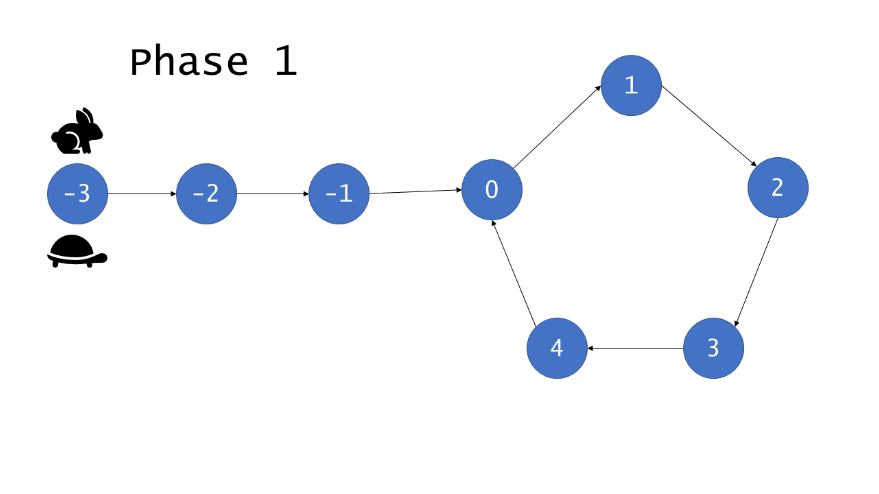
We can harness the fact that hare moves twice as quickly as tortoise to assert that when hare and tortoise meet at node *h*, hare has traversed twice as many nodes. Using this fact, we deduce the following:

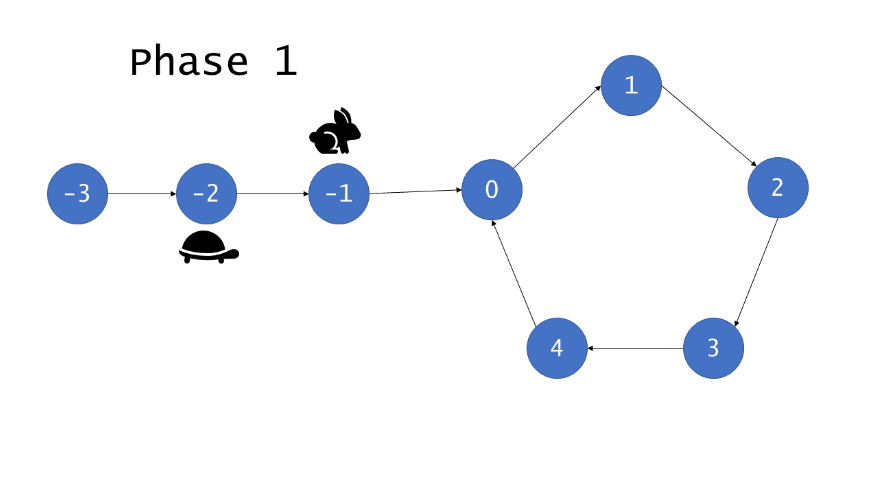
To compute the intersection point, let's note that the hare has traversed twice as many nodes as the tortoise, *i.e.* 2*d*(tortoise)=*d*(hare), that means

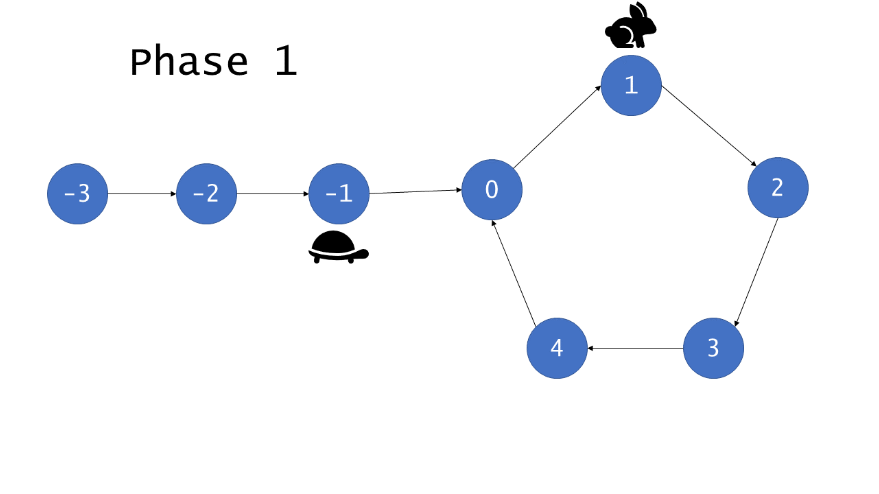


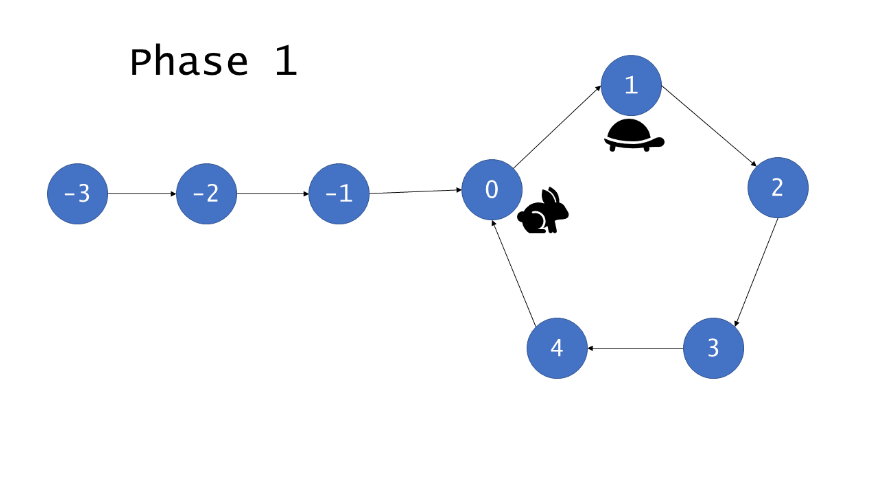
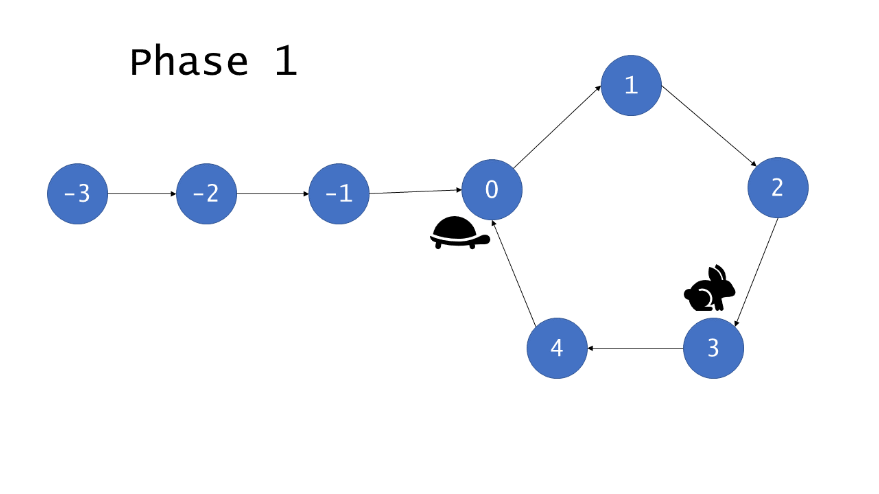
To see the entire algorithm in action, check out the animation below:

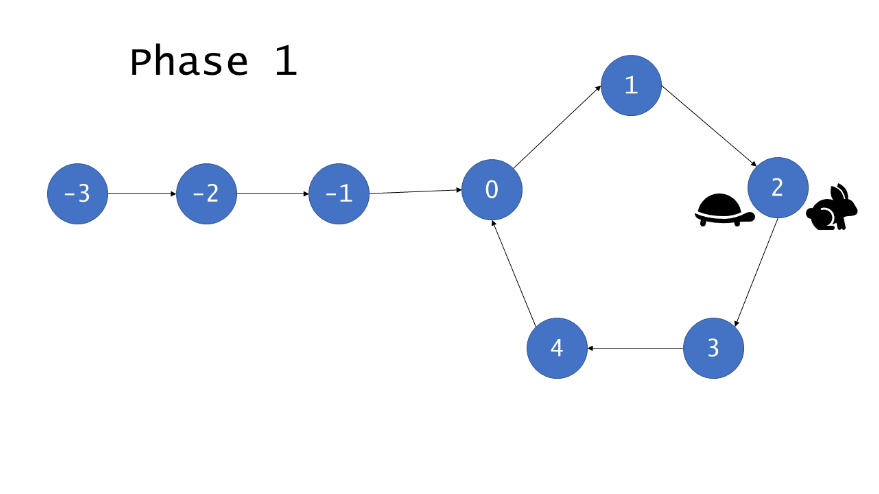


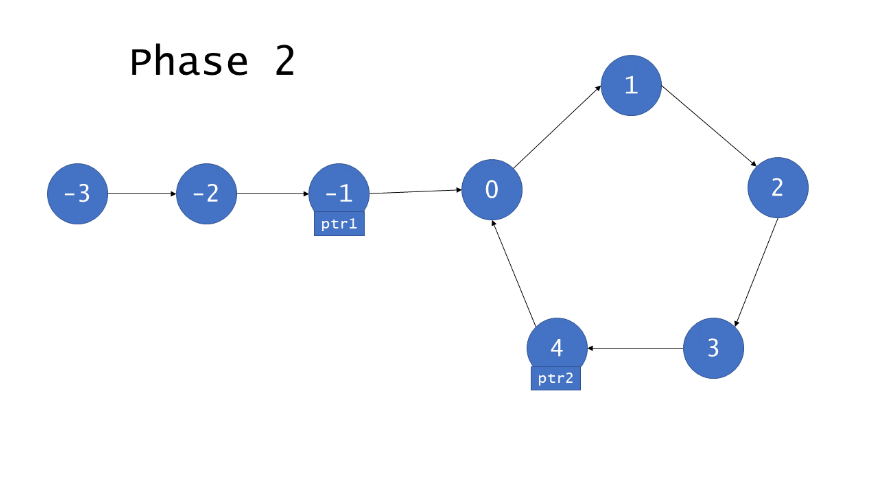
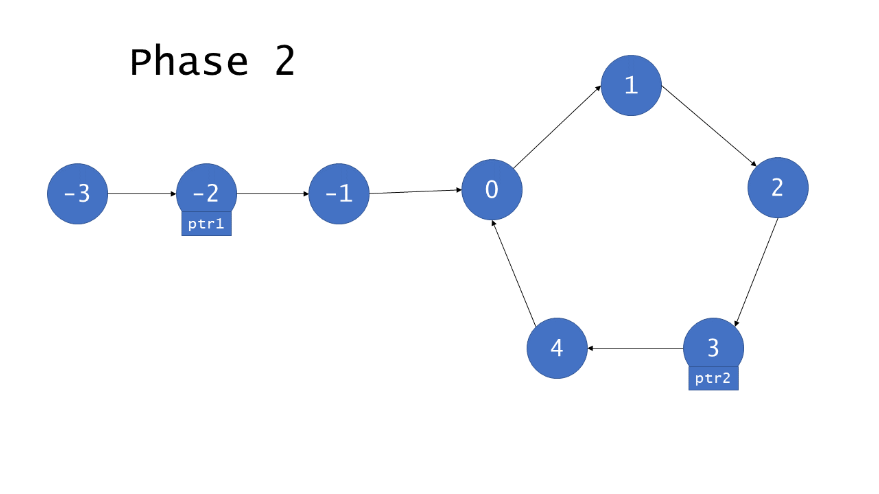
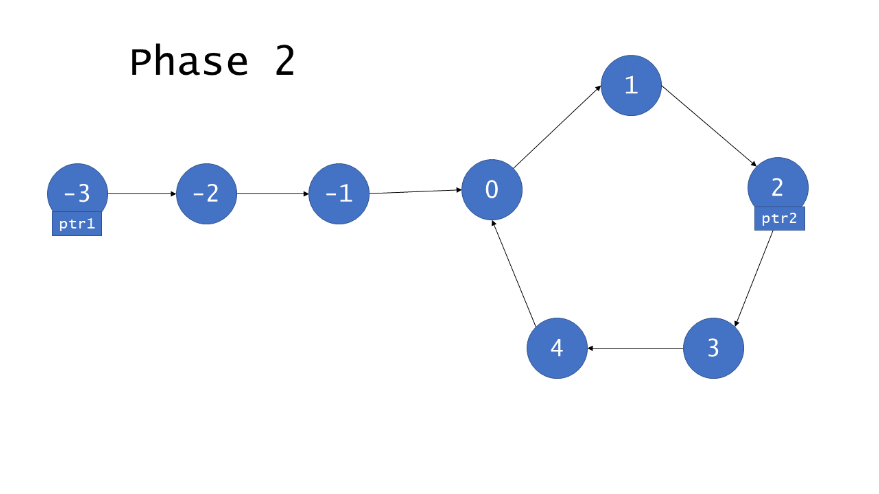
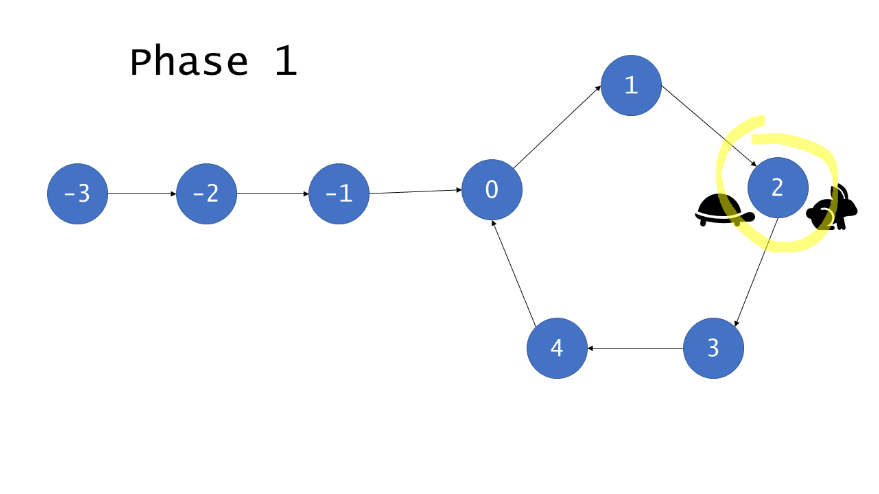


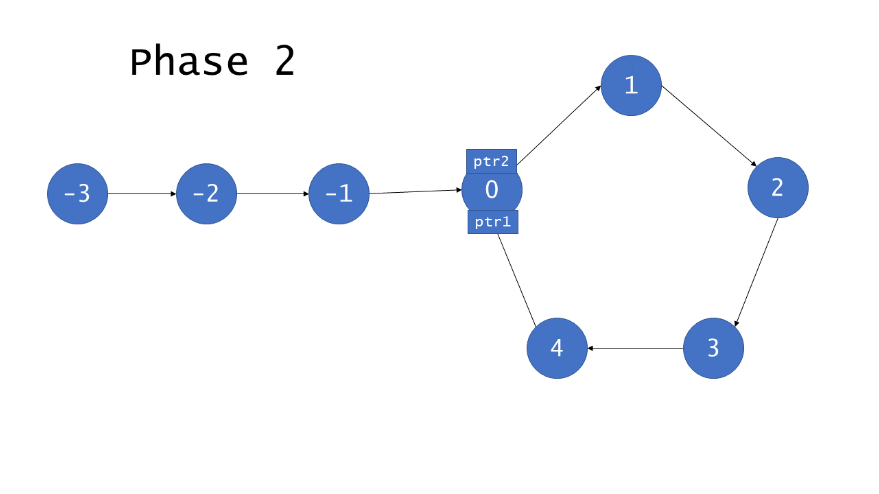


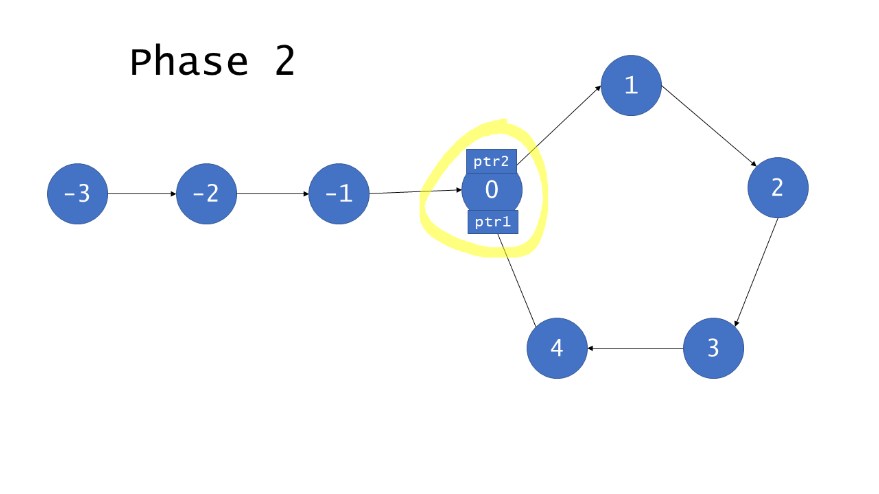












|  |
| --- |
| public class Solution {  private ListNode getIntersect(ListNode head) {  ListNode tortoise = head;  ListNode hare = head;  // A fast pointer will either loop around a cycle and meet the slow  // pointer or reach the `null` at the end of a non-cyclic list.  while (hare != null && hare.next != null) {  tortoise = tortoise.next;  hare = hare.next.next;  if (tortoise == hare) {  return tortoise;  }  }  return null;  }  public ListNode detectCycle(ListNode head) {  if (head == null) {  return null;  }  // If there is a cycle, the fast/slow pointers will intersect at some  // node. Otherwise, there is no cycle, so we cannot find an entrance to  // a cycle.  ListNode intersect = getIntersect(head);  if (intersect == null) {  return null;  }  // To find the entrance to the cycle, we have two pointers traverse at  // the same speed -- one from the front of the list, and the other from  // the point of intersection.  ListNode ptr1 = head;  ListNode ptr2 = intersect;  while (ptr1 != ptr2) {  ptr1 = ptr1.next;  ptr2 = ptr2.next;  }  return ptr1;  }  } |

**Complexity Analysis**

* Time complexity : *O*(*n*)

For cyclic lists, hare and tortoise will point to the same node after *F*+*C*−*h* iterations, as demonstrated in the proof of correctness. *F*+*C*−*h* ≤ *F*+*C*=*n*, so phase 1 runs in *O*(*n*) time. Phase 2 runs for *F*<*n* iterations, so it also runs in *O*(*n*) time.

For acyclic lists, hare will reach the end of the list in roughly n/2​ iterations, causing the function to return before phase 2. Therefore, regardless of which category of list the algorithm receives, it runs in time linearly proportional to the number of nodes.

* Space complexity : *O*(1)

Floyd's Tortoise and Hare algorithm allocates only pointers, so it runs with constant overall memory usage.

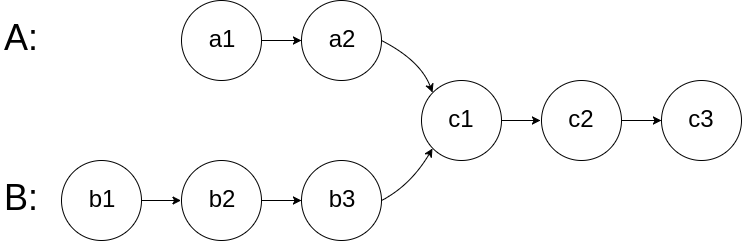
**Footnotes**

1. It is sufficient to check only hare because it will always be ahead of tortoise in an acyclic list. [↩︎](https://leetcode.com/problems/linked-list-cycle-ii/solution/#fnref1)

**Intersection of Two Linked Lists**

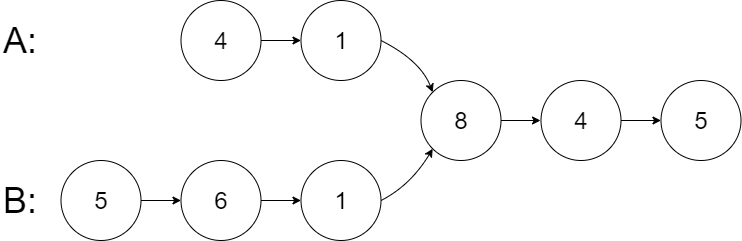
Write a program to find the node at which the intersection of two singly linked lists begins.

For example, the following two linked lists:



begin to intersect at node c1.

**Example 1:**

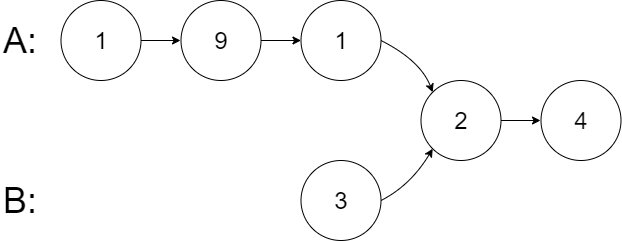


**Input:** intersectVal = 8, listA = [4,1,8,4,5], listB = [5,6,1,8,4,5], skipA = 2, skipB = 3

**Output:** Reference of the node with value = 8

**Input Explanation:** The intersected node's value is 8 (note that this must not be 0 if the two lists intersect). From the head of A, it reads as [4,1,8,4,5]. From the head of B, it reads as [5,6,1,8,4,5]. There are 2 nodes before the intersected node in A; There are 3 nodes before the intersected node in B.

**Example 2:**

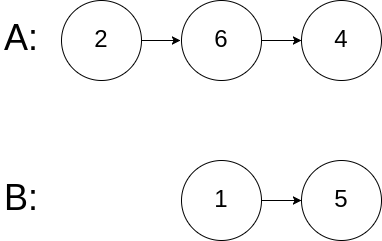


**Input:** intersectVal = 2, listA = [1,9,1,2,4], listB = [3,2,4], skipA = 3, skipB = 1

**Output:** Reference of the node with value = 2

**Input Explanation:** The intersected node's value is 2 (note that this must not be 0 if the two lists intersect). From the head of A, it reads as [1,9,1,2,4]. From the head of B, it reads as [3,2,4]. There are 3 nodes before the intersected node in A; There are 1 node before the intersected node in B.

**Example 3:**



**Input:** intersectVal = 0, listA = [2,6,4], listB = [1,5], skipA = 3, skipB = 2

**Output:** null

**Input Explanation:** From the head of A, it reads as [2,6,4]. From the head of B, it reads as [1,5]. Since the two lists do not intersect, intersectVal must be 0, while skipA and skipB can be arbitrary values.

**Explanation:** The two lists do not intersect, so return null.

**Notes:**

* If the two linked lists have no intersection at all, return null.
* The linked lists must retain their original structure after the function returns.
* You may assume there are no cycles anywhere in the entire linked structure.
* Each value on each linked list is in the range [1, 10^9].
* Your code should preferably run in O(n) time and use only O(1) memory.

## Solution

#### **Approach 1: Brute Force**

For each node ai in list A, traverse the entire list B and check if any node in list B coincides with ai.

**Complexity Analysis**

* Time complexity : *O*(*mn*).
* Space complexity : *O*(1).

#### **Approach 2: Hash Table**

Traverse list A and store the address / reference to each node in a hash set. Then check every node bi in list B: if bi appears in the hash set, then bi is the intersection node.

**Complexity Analysis**

* Time complexity : *O*(*m*+*n*).
* Space complexity : *O*(*m*) or *O*(*n*).

#### **Approach 3: Two Pointers**

* Maintain two pointers *pA* and *pB* initialized at the head of A and B, respectively. Then let them both traverse through the lists, one node at a time.
* When *pA* reaches the end of a list, then redirect it to the head of B (yes, B, that's right.); similarly when *pB* reaches the end of a list, redirect it the head of A.
* If at any point *pA* meets *pB*, then *pA*/*pB* is the intersection node.
* To see why the above trick would work, consider the following two lists: A = {1,3,5,7,9,11} and B = {2,4,9,11}, which are intersected at node '9'. Since B.length (=4) < A.length (=6), *pB* would reach the end of the merged list first, because *pB* traverses exactly 2 nodes less than *pA* does. By redirecting *pB* to head A, and *pA* to head B, we now ask *pB* to travel exactly 2 more nodes than *pA* would. So in the second iteration, they are guaranteed to reach the intersection node at the same time.
* If two lists have intersection, then their last nodes must be the same one. So when *pA*/*pB* reaches the end of a list, record the last element of A/B respectively. If the two last elements are not the same one, then the two lists have no intersections.

**Complexity Analysis**

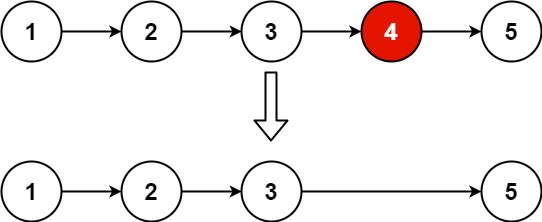
* Time complexity : *O*(*m*+*n*).
* Space complexity : *O*(1).

**Remove Nth Node From End of List**

Given the head of a linked list, remove the nth node from the end of the list and return its head.

**Follow up:** Could you do this in one pass?

**Example 1:**



**Input:** head = [1,2,3,4,5], n = 2

**Output:** [1,2,3,5]

**Example 2:**

**Input:** head = [1], n = 1

**Output:** []

**Example 3:**

**Input:** head = [1,2], n = 1

**Output:** [1]

**Constraints:**

* The number of nodes in the list is sz.
* 1 <= sz <= 30
* 0 <= Node.val <= 100
* 1 <= n <= sz

Hint #1

Maintain two pointers and update one with a delay of n steps.

## Summary

This article is for beginners. It introduces the following idea: Linked List traversal and removal of nth element from the end.

## Solution

#### **Approach 1: Two pass algorithm**

**Intuition**

We notice that the problem could be simply reduced to another one : Remove the (*L*−*n*+1) th node from the beginning in the list , where *L* is the list length. This problem is easy to solve once we found list length *L*.

**Algorithm**

First we will add an auxiliary "dummy" node, which points to the list head. The "dummy" node is used to simplify some corner cases such as a list with only one node, or removing the head of the list. On the first pass, we find the list length *L*. Then we set a pointer to the dummy node and start to move it through the list till it comes to the (*L*−*n*) th node. We relink next pointer of the (*L*−*n*) th node to the (*L*−*n*+2) th node and we are done.

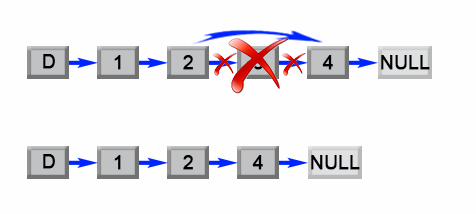


Figure 1. Remove the L - n + 1 th element from a list.

|  |
| --- |
| public ListNode removeNthFromEnd(ListNode head, int n) {  ListNode dummy = new ListNode(0);  dummy.next = head;  int length = 0;  ListNode first = head;  while (first != null) {  length++;  first = first.next;  }  length -= n;  first = dummy;  while (length > 0) {  length--;  first = first.next;  }  first.next = first.next.next;  return dummy.next;  } |

**Complexity Analysis**

* Time complexity : *O*(*L*).

The algorithm makes two traversal of the list, first to calculate list length *L* and second to find the (*L*−*n*) th node. There are 2*L*−*n* operations and time complexity is *O*(*L*).

* Space complexity : *O*(1).

We only used constant extra space.

#### **Approach 2: One pass algorithm**

**Algorithm**

The above algorithm could be optimized to one pass. Instead of one pointer, we could use two pointers. The first pointer advances the list by *n*+1 steps from the beginning, while the second pointer starts from the beginning of the list. Now, both pointers are exactly separated by *n* nodes apart. We maintain this constant gap by advancing both pointers together until the first pointer arrives past the last node. The second pointer will be pointing at the *n*th node counting from the last. We relink the next pointer of the node referenced by the second pointer to point to the node's next next node.

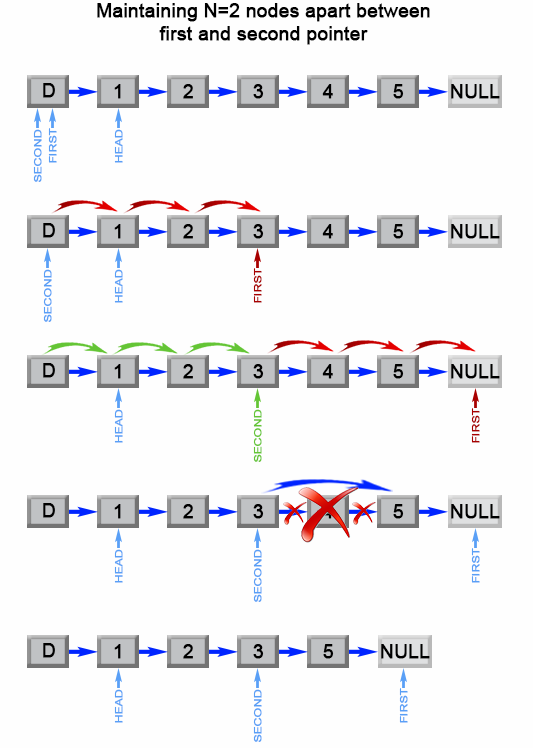


Figure 2. Remove the nth element from end of a list.

|  |
| --- |
| public ListNode removeNthFromEnd(ListNode head, int n) {  ListNode dummy = new ListNode(0);  dummy.next = head;  ListNode first = dummy;  ListNode second = dummy;  // Advances first pointer so that the gap between first and second is n nodes apart  for (int i = 1; i <= n + 1; i++) {  first = first.next;  }  // Move first to the end, maintaining the gap  while (first != null) {  first = first.next;  second = second.next;  }  second.next = second.next.next;  return dummy.next;  } |

**Complexity Analysis**

* Time complexity : *O*(*L*).

The algorithm makes one traversal of the list of *L* nodes. Therefore time complexity is *O*(*L*).

* Space complexity : *O*(1).

We only used constant extra space.

**Summary - Two-Pointer in Linked List**

Here we provide a template for you to solve the two-pointer problem in the linked list.

|  |
| --- |
| // Initialize slow & fast pointers  ListNode slow = head;  ListNode fast = head;  /\*\*  \* Change this condition to fit specific problem.  \* Attention: remember to avoid null-pointer error  \*\*/  while (slow != null && fast != null && fast.next != null) {  slow = slow.next; // move slow pointer one step each time  fast = fast.next.next; // move fast pointer two steps each time  if (slow == fast) { // change this condition to fit specific problem  return true;  }  }  return false; // change return value to fit specific problem |

### ***Tips***

It is similar to what we have learned in an array. But it can be trickier and error-prone. There are several things you should pay attention:

**1. Always examine if the node is null before you call the next field.**

Getting the next node of a null node will cause the null-pointer error. For example, before we run fast = fast.next.next, we need to examine both fast and fast.next is not null.

**2. Carefully define the end conditions of your loop.**

Run several examples to make sure your end conditions will not result in an endless loop. And you have to take our first tip into consideration when you define your end conditions.

### ***Complexity Analysis***

It is easy to analyze the space complexity. If you only use pointers without any other extra space, the space complexity will be O(1). However, it is more difficult to analyze the time complexity. In order to get the answer, we need to analyze how many times we will run our loop .

In our previous finding cycle example, let's assume that we move the faster pointer 2 steps each time and move the slower pointer 1 step each time.

1. If there is no cycle, the fast pointer takes N/2 times to reach the end of the linked list, where N is the length of the linked list.
2. If there is a cycle, the fast pointer needs M times to catch up the slower pointer, where M is the length of the cycle in the list.

Obviously, M <= N. So we will run the loop up to N times. And for each loop, we only need constant time. So, the time complexity of this algorithm is O(N) in total.

Analyze other problems by yourself to improve your analysis skill. Don't forget to take different conditions into consideration. If it is hard to analyze for all situations, consider the worst one.

## Classic Problems

In the last chapter, we have introduced how to use the two-pointer technique in a linked list. In this chapter, we will start with how to reverse a singly linked list and explore more classic problems.

**Reverse Linked List**

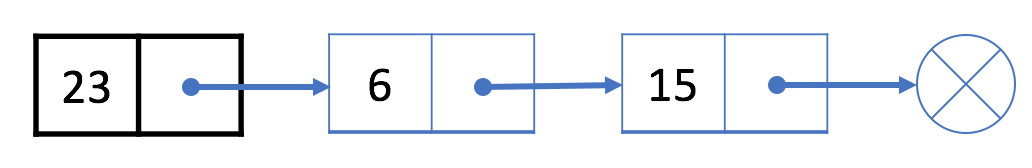
Let's start with a classic problem:

Reverse a singly linked list.

One solution is to iterate the nodes in original order and move them to the head of the list one by one. It seems hard to understand. We will first use an example to go through our algorithm.

### ***Algorithm Overview***

Let's look at an example:

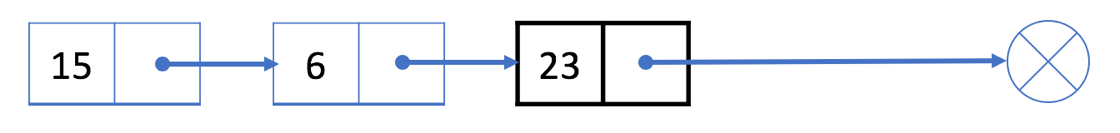


Keep in mind that the black node 23 is our original head node.

1. First, we move the next node of the black node, which is node 6, to the head of the list:



2. Then we move the next node of the black node, which is node 15, to the head of the list:



3. The next node of the black node now is null. So we stop and return our new head node 15.

### ***More***

In this algorithm, each node will be moved exactly once.

Therefore, the time complexity is O(N), where N is the length of the linked list. We only use constant extra space so the space complexity is O(1).

This problem is the foundation of many linked-list problems you might come across in your interview. If you are still stuck, our next article will talk more about the implementation details.

There are also many other solutions. You should be familiar with at least one solution and be able to implement it.

**Reverse Linked List**

Reverse a singly linked list.

**Example:**

**Input:** 1->2->3->4->5->NULL

**Output:** 5->4->3->2->1->NULL

**Follow up:**

A linked list can be reversed either iteratively or recursively. Could you implement both?

## Solution

#### **Approach #1 (Iterative) [Accepted]**

Assume that we have linked list 1 → 2 → 3 → Ø, we would like to change it to Ø ← 1 ← 2 ← 3.

While you are traversing the list, change the current node's next pointer to point to its previous element. Since a node does not have reference to its previous node, you must store its previous element beforehand. You also need another pointer to store the next node before changing the reference. Do not forget to return the new head reference at the end!

|  |
| --- |
| public ListNode reverseList(ListNode head) {  ListNode prev = null;  ListNode curr = head;  while (curr != null) {  ListNode nextTemp = curr.next;  curr.next = prev;  prev = curr;  curr = nextTemp;  }  return prev;  } |

**Complexity analysis**

* Time complexity : *O*(*n*). Assume that *n* is the list's length, the time complexity is *O*(*n*).
* Space complexity : *O*(1).

#### **Approach #2 (Recursive) [Accepted]**

The recursive version is slightly trickier and the key is to work backwards. Assume that the rest of the list had already been reversed, now how do I reverse the front part? Let's assume the list is: n1 → … → nk-1 → nk → nk+1 → … → nm → Ø

Assume from node nk+1 to nm had been reversed and you are at node nk.

n1 → … → nk-1 → **nk** → nk+1 ← … ← nm

We want nk+1’s next node to point to nk.

So,

nk.next.next = nk;

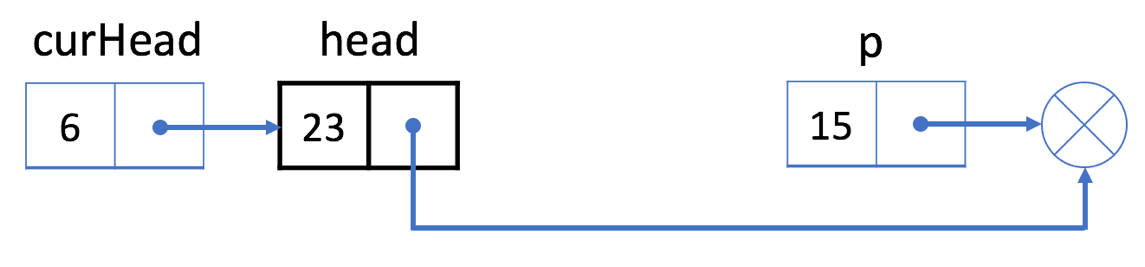
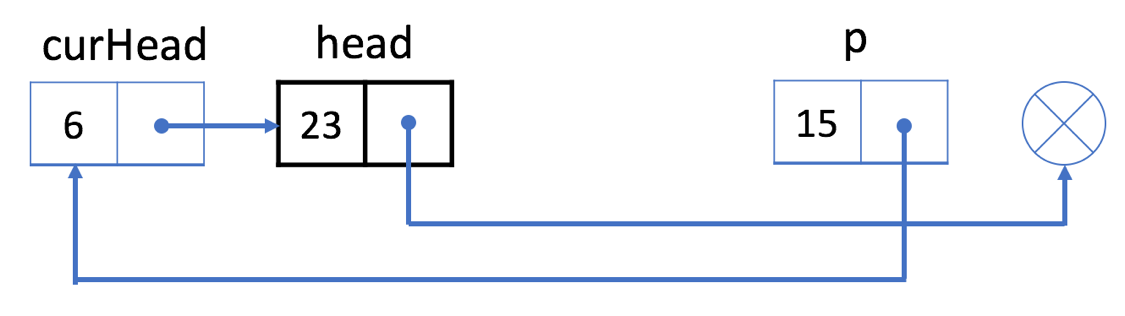
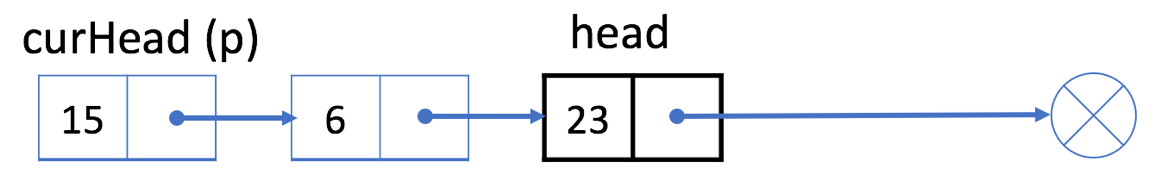
Be very careful that n1's next must point to Ø. If you forget about this, your linked list has a cycle in it. This bug could be caught if you test your code with a linked list of size 2.

|  |
| --- |
| public ListNode reverseList(ListNode head) {  if (head == null || head.next == null) return head;  ListNode p = reverseList(head.next);  head.next.next = head;  head.next = null;  return p;  } |

**Complexity analysis**

* Time complexity : *O*(*n*). Assume that *n* is the list's length, the time complexity is *O*(*n*).
* Space complexity : *O*(*n*). The extra space comes from implicit stack space due to recursion. The recursion could go up to *n* levels deep.

**Reverse Linked List - Solution**

* In this article, we will talk more about details of our algorithm to reverse the linked list.
* In the solution we mentioned previously, there are two nodes which we should keep track of: the original head node and the new head node.
* Therefore, we need to use two pointers in one linked list at the same time. One pointer head always points at our original head node while another pointer curHead always points at our newest head node.
* Let's focus on a single step (the 2nd step in the [previous article](https://leetcode.com/explore/learn/card/linked-list/219/linked-list-classic-problem/1204/)). Our goal is to move the next node of head, which is 15, to the head of the list.
* 
* 1. First, we use a temporary pointer p to indicate the next node of the head node. And link the "next" field of head to the "next" field of p.
* 
* 2. Then, we link the "next" field of p to the curHead.
* 
* 3. Now our linked list actually looks like the picture below. And we set curHead to be p.
* 
* By this way, we successfully move node 15 to the head of the list. And we can repeat this process until the next node of head is null.

### ***Reference Code***

* Here we provide code in different languages for your reference:

|  |
| --- |
| /\*\*  \* Definition for singly-linked list.  \* public class ListNode {  \* int val;  \* ListNode next;  \* ListNode(int x) { val = x; }  \* }  \*/  class Solution {  public ListNode reverseList(ListNode head) {  if (head == null) {  return head;  }  ListNode currentHead = head;  while (head.next != null) {  ListNode p = head.next;  head.next = p.next;  p.next = currentHead;  currentHead = p;  }  return currentHead;  }  } |

**Remove Linked List Elements**

Remove all elements from a linked list of integers that have value ***val***.

**Example:**

**Input:** 1->2->6->3->4->5->6, ***val*** = 6

**Output:** 1->2->3->4->5

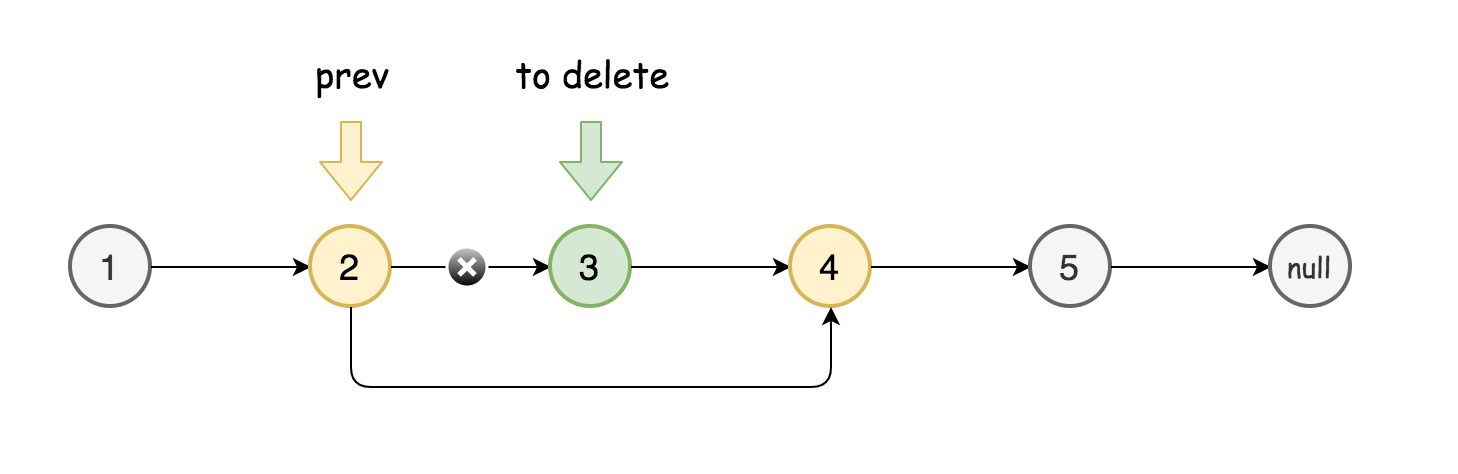
## Solution

#### **Approach 1: Sentinel Node**

**Intuition**

The problem seems to be very easy if one has to delete a node in the middle:

* Pick the node-predecessor prev of the node to delete.
* Set its next pointer to point to the node next to the one to delete.



The things are more complicated when the node or nodes to delete are in the head of linked list.

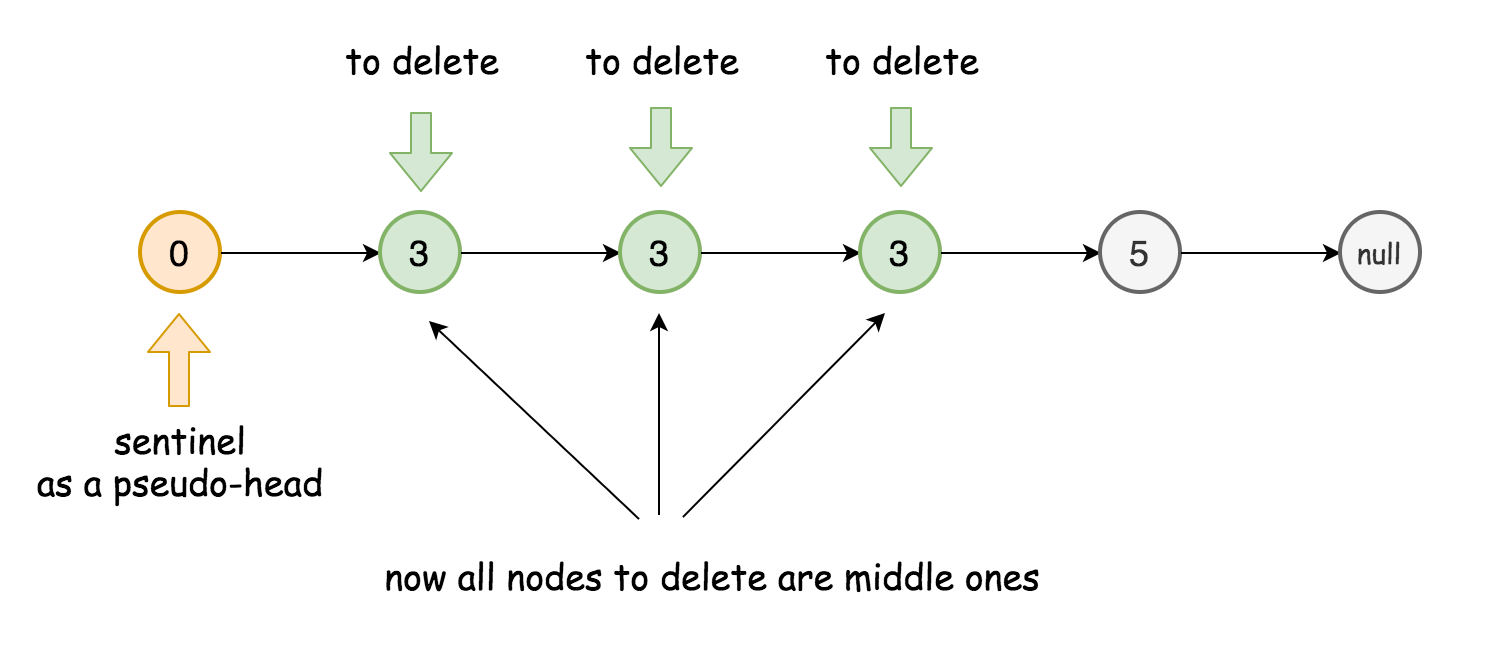


How to deal with that? To reduce the problem to the deletion of middle nodes with the help of [sentinel node](https://en.wikipedia.org/wiki/Sentinel_node).

Sentinel nodes are widely used in trees and linked lists as pseudo-heads, pseudo-tails, markers of level end, etc. They are purely functional, and usually does not hold any data. Their main purpose is to standardize the situation, for example, make linked list to be never empty and never headless and hence simplify insert and delete.

Here are two standard examples:

* [LRU Cache](https://leetcode.com/articles/lru-cache/), sentinel nodes are used as pseudo-head and pseudo-tail.
* [Tree Level Order Traversal](https://leetcode.com/articles/maximum-level-sum-of-a-binary-tree/), sentinel nodes are used to mark level end.



Here sentinel node will be used as pseudo-head.

**Algorithm**

* Initiate sentinel node as ListNode(0) and set it to be the new head: sentinel.next = head.
* Initiate two pointers to track the current node and its predecessor: curr and prev.
* While curr is not a null pointer:
  + Compare the value of the current node with the value to delete.
    - In the values are equal, delete the current node: prev.next = curr.next.
    - Otherwise, set predecessor to be equal to the current node.
  + Move to the next node: curr = curr.next.
* Return sentinel.next.

**Implementation**

|  |
| --- |
| class Solution {  public ListNode removeElements(ListNode head, int val) {  ListNode sentinel = new ListNode(0);  sentinel.next = head;  ListNode prev = sentinel, curr = head;  while (curr != null) {  if (curr.val == val) prev.next = curr.next;  else prev = curr;  curr = curr.next;  }  return sentinel.next;  }  } |

**Complexity Analysis**

* Time complexity: O(*N*), it's one pass solution.
* Space complexity: O(1), it's a constant space solution.

**Odd Even Linked List**

Given a singly linked list, group all odd nodes together followed by the even nodes. Please note here we are talking about the node number and not the value in the nodes.

You should try to do it in place. The program should run in O(1) space complexity and O(nodes) time complexity.

**Example 1:**

**Input:** 1->2->3->4->5->NULL

**Output:** 1->3->5->2->4->NULL

**Example 2:**

**Input:** 2->1->3->5->6->4->7->NULL

**Output:** 2->3->6->7->1->5->4->NULL

**Constraints:**

* The relative order inside both the even and odd groups should remain as it was in the input.
* The first node is considered odd, the second node even and so on ...
* The length of the linked list is between [0, 10^4].

## Solution

**Intuition**

Put the odd nodes in a linked list and the even nodes in another. Then link the evenList to the tail of the oddList.

**Algorithm**

The solution is very intuitive. But it is not trivial to write a concise and bug-free code.

A well-formed LinkedList need two pointers head and tail to support operations at both ends. The variables head and odd are the head pointer and tail pointer of one LinkedList we call oddList; the variables evenHead and even are the head pointer and tail pointer of another LinkedList we call evenList. The algorithm traverses the original LinkedList and put the odd nodes into the oddList and the even nodes into the evenList. To traverse a LinkedList we need at least one pointer as an iterator for the current node. But here the pointers odd and even not only serve as the tail pointers but also act as the iterators of the original list.

The best way of solving any linked list problem is to visualize it either in your mind or on a piece of paper. An illustration of our algorithm is following:

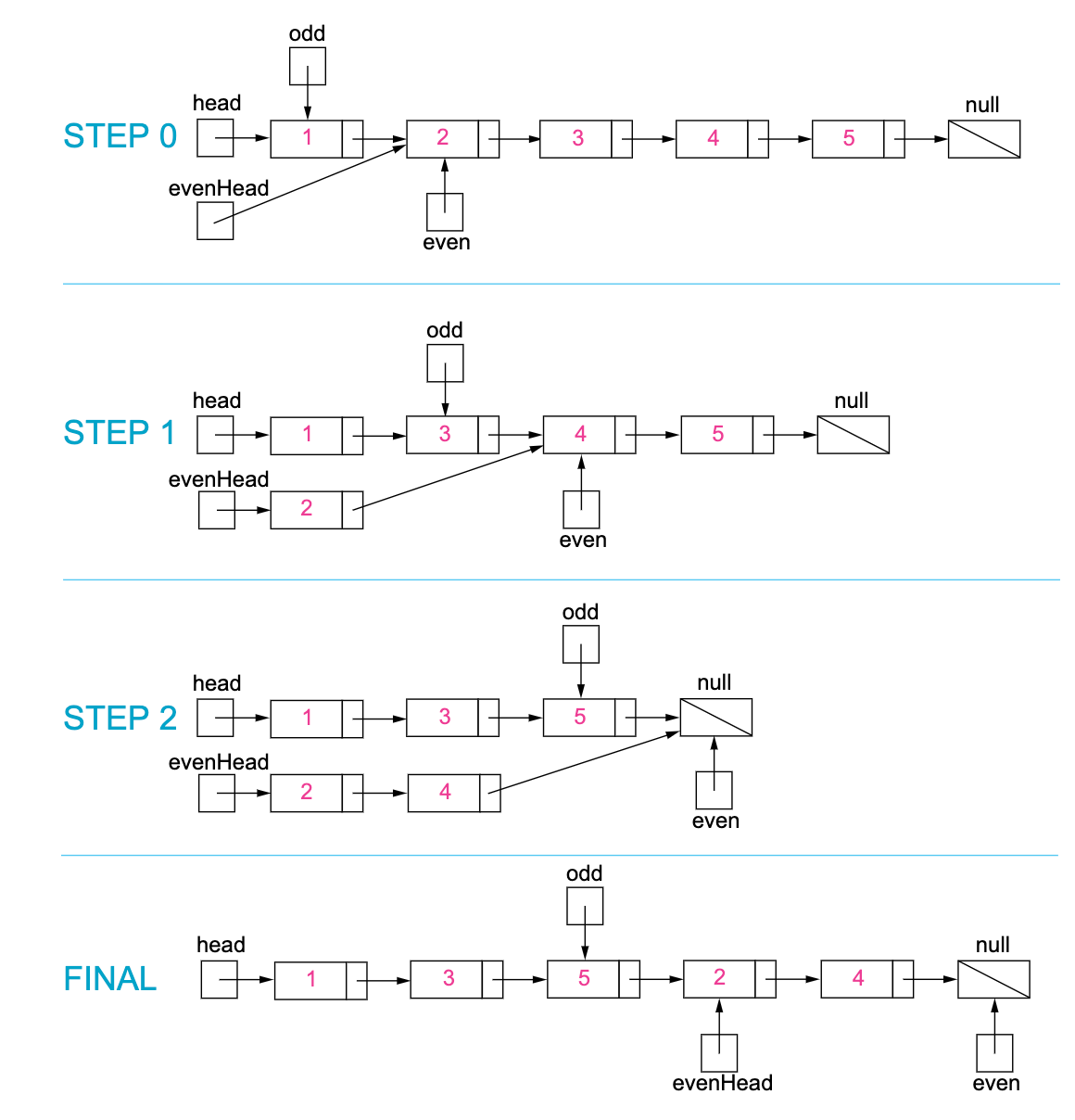


Figure 1. Step by step example of the odd and even linked list.

|  |
| --- |
| public class Solution {  public ListNode oddEvenList(ListNode head) {  if (head == null) return null;  ListNode odd = head, even = head.next, evenHead = even;  while (even != null && even.next != null) {  odd.next = even.next;  odd = odd.next;  even.next = odd.next;  even = even.next;  }  odd.next = evenHead;  return head;  }  } |

**Complexity Analysis**

* Time complexity : *O*(*n*). There are total *n* nodes and we visit each node once.
* Space complexity : *O*(1). All we need is the four pointers.

**Palindrome Linked List**

Given a singly linked list, determine if it is a palindrome.

**Example 1:**

**Input:** 1->2

**Output:** false

**Example 2:**

**Input:** 1->2->2->1

**Output:** true

**Follow up:**  
Could you do it in O(n) time and O(1) space?

## Solution

#### **Approach 1: Copy into Array List and then Use Two Pointer Technique**

**Intuition**

If you're not too familiar with **Linked Lists** yet, here's a quick recap on **Lists**.

There are two commonly used **List** implementations, the **Array List** and the **Linked List**. If we have some values we want to store in a list, how would each List implementation hold them?

* An **Array List** uses an underlying **Array** to store the values. We can access the value at any position in the list in *O*(1) time, as long as we know the index. This is based on the underlying memory addressing.
* A **Linked List** uses **Objects** commonly called **Nodes**. Each **Node** holds a value and a next pointer to the next node. Accessing a node at a particular index would take *O*(*n*) time because we have to go down the list using the next pointers.

Determining whether or not an Array List is a palindrome is straightforward. We can simply use the **two-pointer technique** to compare indexes at either end, moving in towards the middle. One pointer starts at the start and goes up, and the other starts at the end and goes down. This would take *O*(*n*) because each index access is *O*(1) and there are *n* index accesses in total.

However, it's not so straightforward for a Linked List. Accessing the values in any order other than forward, sequentially, is ***not*** *O*(1). Unfortunately, this includes (iteratively) accessing the values in reverse. We will need a completely different approach.

On the plus side, making a copy of the Linked List values into an Array List is *O*(*n*). Therefore, the simplest solution is to copy the values of the Linked List into an Array List (or Vector, or plain Array). Then, we can solve the problem using the **two-pointer technique**.

**Algorithm**

We can split this approach into 2 steps:

1. Copying the Linked List into an Array.
2. Checking whether or not the Array is a palindrome.

To do the first step, we need to iterate through the Linked List, adding each value to an Array. We do this by using a variable currentNode to point at the current Node we're looking at, and at each iteration adding currentNode.val to the array and updating currentNode to point to currentNode.next. We should stop looping once currentNode points to a null node.

The best way of doing the second step depends on the programming language you're using. In Python, it's straightforward to make a reversed copy of a list and also straightforward to compare two lists. In other languages, this is not so straightforward and so it's probably best to use the **two-pointer technique** to check for a palindrome. In the two-pointer technique, we put a pointer at the start and a pointer at the end, and at each step check the values are equal and then move the pointers inwards until they meet at the center.

When writing your own solution, remember that we need to compare values in the nodes, not the nodes themselves. node\_1.val == node\_2.val is the correct way of comparing the nodes. node\_1 == node\_2 will not work the way you expect.

|  |
| --- |
| class Solution {  public boolean isPalindrome(ListNode head) {  List<Integer> vals = new ArrayList<>();  // Convert LinkedList into ArrayList.  ListNode currentNode = head;  while (currentNode != null) {  vals.add(currentNode.val);  currentNode = currentNode.next;  }  // Use two-pointer technique to check for palindrome.  int front = 0;  int back = vals.size() - 1;  while (front < back) {  // Note that we must use ! .equals instead of !=  // because we are comparing Integer, not int.  if (!vals.get(front).equals(vals.get(back))) {  return false;  }  front++;  back--;  }  return true;  }  } |

**Complexity Analysis**

* Time complexity : *O*(*n*), where *n* is the number of nodes in the Linked List.

In the first part, we're copying a Linked List into an Array List. Iterating down a Linked List in sequential order is *O*(*n*), and each of the *n* writes to the ArrayList is *O*(1). Therefore, the overall cost is *O*(*n*).

In the second part, we're using the two pointer technique to check whether or not the Array List is a palindrome. Each of the *n* values in the Array list is accessed once, and a total of *O*(*n*/2) comparisons are done. Therefore, the overall cost is *O*(*n*). The Python trick (reverse and list comparison as a one liner) is also *O*(*n*).

This gives *O*(2*n*)=*O*(*n*) because we always drop the constants.

* Space complexity : *O*(*n*), where *n* is the number of nodes in the Linked List.

We are making an Array List and adding *n* values to it.

#### **Approach 2: Recursive (Advanced)**

**Intuition**

In an attempt to come up with a way of using *O*(1) space, you might have thought of using recursion. However, this is still *O*(*n*) space. Let's have a look at it and understand why it is **not** *O*(1) space.

Recursion gives us an elegant way to iterate through the nodes in reverse. For example, this algorithm will print out the values of the nodes in reverse. Given a node, the algorithm checks if it is null. If it is null, nothing happens. Otherwise, all nodes after it are processed, and then the value for the current node is printed.

function print\_values\_in\_reverse(ListNode head)

if head is NOT null

print\_values\_in\_reverse(head.next)

print head.val

If we iterate the nodes in reverse using recursion, and iterate forward at the same time using a variable outside the recursive function, then we can check whether or not we have a palindrome.

**Algorithm**

When given the head node (or any other node), referred to as currentNode, the algorithm firstly checks the rest of the Linked List. If it discovers that further down that the Linked List is not a palindrome, then it returns false. Otherwise, it checks that currentNode.val == frontPointer.val. If not, then it returns false. Otherwise, it moves frontPointer forward by 1 node and returns true to say that from this point forward, the Linked List is a valid palindrome.

It might initially seem surprisingly that frontPointer is always pointing where we want it. The reason it works is because the order in which nodes are processed by the recursion is in reverse (remember our "printing" algorithm above). Each node compares itself against frontPointer and then moves frontPointer down by 1, ready for the next node in the recursion. In essence, we are iterating both backwards and forwards at the same time.

Here is an animation that shows how the algorithm works. The nodes have each been given a unique identifier (e.g. 1` and `1‘*and*‘4) so that they can more easily be referred to in the explanations. The computer needs to use its runtime stack for recursive functions.

|  |
| --- |
| class Solution {  private ListNode frontPointer;  private boolean recursivelyCheck(ListNode currentNode) {  if (currentNode != null) {  if (!recursivelyCheck(currentNode.next)) return false;  if (currentNode.val != frontPointer.val) return false;  frontPointer = frontPointer.next;  }  return true;  }  public boolean isPalindrome(ListNode head) {  frontPointer = head;  return recursivelyCheck(head);  }  } |

**Complexity Analysis**

* Time complexity : *O*(*n*), where *n* is the number of nodes in the Linked List.

The recursive function is run once for each of the *n* nodes, and the body of the recursive function is *O*(1). Therefore, this gives a total of *O*(*n*).

* Space complexity : *O*(*n*), where *n* is the number of nodes in the Linked List.

I hinted at the start that this is not using *O*(1) space. This might seem strange, after all we aren't creating any new data structures. So, where is the *O*(*n*) extra memory we're using? Understanding what is happening here requires understanding how the computer runs a recursive function.

Each time a function is called within a function, the computer needs to keep track of where it is up to (and the values of any local variables) in the current function before it goes into the called function. It does this by putting an entry on something called the **runtime stack**, called a **stack frame**. Once it has created a stack frame for the current function, it can then go into the called function. Then once it is finished with the called function, it pops the top stack frame to resume the function it had been in before the function call was made.

Before doing any palindrome checking, the above recursive function creates *n* of these stack frames because the first step of processing a node is to process the nodes after it, which is done with a recursive call. Then once it has the *n* stack frames, it pops them off one-by-one to process them.

So, the space usage is on the runtime stack because we are creating *n* stack frames. Always make sure to consider what's going on the runtime stack when analyzing a recursive function. It's a common mistake to forget to.

Not only is this approach still using *O*(*n*) space, it is worse than the first approach because in many languages (such as Python), stack frames are large, and there's a maximum runtime stack depth of 1000 (you can increase it, but you risk causing memory errors with the underlying interpreter). With every node creating a stack frame, this will greatly limit the maximum Linked List size the algorithm can handle.

#### **Approach 3: Reverse Second Half In-place**

**Intuition**

The ***only*** way we can avoid using *O*(*n*) extra space is by modifying the input in-place.

The strategy we can use is to reverse the second half of the Linked List in-place (modifying the Linked List structure), and then comparing it with the first half. Afterwards, we should re-reverse the second half and put the list back together. While you don't need to restore the list to pass the test cases, it is still good programming practice because the function could be a part of a bigger program that doesn't want the Linked List broken.

**Algorithm**

Specifically, the steps we need to do are:

1. Find the end of the first half.
2. Reverse the second half.
3. Determine whether or not there is a palindrome.
4. Restore the list.
5. Return the result.

To do step 1, we could count the number of nodes, calculate how many nodes are in the first half, and then iterate back down the list to find the end of the first half. Or, we could do it in a single parse using the **two runners pointer technique**. Either is acceptable, however we'll have a look at the two runners pointer technique here.

Imagine we have 2 runners one fast and one slow, running down the nodes of the Linked List. In each second, the fast runner moves down 2 nodes, and the slow runner just 1 node. By the time the fast runner gets to the end of the list, the slow runner will be half way. By representing the runners as pointers, and moving them down the list at the corresponding speeds, we can use this trick to find the middle of the list, and then split the list into two halves.

If there is an odd-number of nodes, then the "middle" node should remain attached to the first half.

Step 2 uses the algorithm that can be found in the solution article for the [Reverse Linked List](https://leetcode.com/problems/reverse-linked-list/) problem to reverse the second half of the list.

Step 3 is fairly straightforward. Remember that we have the first half, which might also contain a "middle" node at the end, and the second half, which is reversed. We can step down the lists simultaneously ensuring the node values are equal. When the node we're up to in the second list is null, we know we're done. If there was a middle value attached to the end of the first list, it is correctly ignored by the algorithm. The result should be saved, but not returned, as we still need to restore the list.

Step 4 requires using the same function you used for step 2, and then for step 5 the saved result should be returned.

|  |
| --- |
| class Solution {  public boolean isPalindrome(ListNode head) {  if (head == null) return true;  // Find the end of first half and reverse second half.  ListNode firstHalfEnd = endOfFirstHalf(head);  ListNode secondHalfStart = reverseList(firstHalfEnd.next);  // Check whether or not there is a palindrome.  ListNode p1 = head;  ListNode p2 = secondHalfStart;  boolean result = true;  while (result && p2 != null) {  if (p1.val != p2.val) result = false;  p1 = p1.next;  p2 = p2.next;  }  // Restore the list and return the result.  firstHalfEnd.next = reverseList(secondHalfStart);  return result;  }  // Taken from https://leetcode.com/problems/reverse-linked-list/solution/  private ListNode reverseList(ListNode head) {  ListNode prev = null;  ListNode curr = head;  while (curr != null) {  ListNode nextTemp = curr.next;  curr.next = prev;  prev = curr;  curr = nextTemp;  }  return prev;  }  private ListNode endOfFirstHalf(ListNode head) {  ListNode fast = head;  ListNode slow = head;  while (fast.next != null && fast.next.next != null) {  fast = fast.next.next;  slow = slow.next;  }  return slow;  }  } |

**Complexity Analysis**

* Time complexity : *O*(*n*), where *n* is the number of nodes in the Linked List.

Similar to the above approaches. Finding the middle is *O*(*n*), reversing a list in place is *O*(*n*), and then comparing the 2 resulting Linked Lists is also *O*(*n*).

* Space complexity : *O*(1).

We are changing the next pointers for half of the nodes. This was all memory that had already been allocated, so we are not using any extra memory and therefore it is *O*(1).

I have seen some people on the discussion forum saying it has to be O(n)*O*(*n*) because we're creating a new list. This is incorrect, because we are changing each of the pointers one-by-one, in-place. We are not needing to allocate more than *O*(1) extra memory to do this work, and there is *O*(1) stack frames going on the stack. It is the same as reversing the values in an Array in place (using the two-pointer technique).

The downside of this approach is that in a concurrent environment (multiple threads and processes accessing the same data), access to the Linked List by other threads or processes would have to be locked while this function is running, because the Linked List is temporarily broken. This is a limitation of many in-place algorithms though.

**Summary - Linked List Classic Problems**

We have provided several exercises for you. You might have noticed the similarities between them. Here we provide some tips for you:

**1. Going through some test cases will save you time.**

It is not easy to debug when using a linked list. Therefore, it is always useful to try several different examples on your own to validate your algorithm before writing code.

**2. Feel free to use several pointers at the same time.**

Sometimes when you design an algorithm for a linked-list problem, there might be several nodes you want to track at the same time. You should keep in mind which nodes you need to track and feel free to use several different pointers to track these nodes at the same time.

If you use several pointers, it will be better to give them suitable names in case you have to debug or review your code in the future.

**3. In many cases, you need to track the previous node of the current node.**

You are not able to trace back the previous node in a singly linked list. So you have to store not only the current node but also the previous node. This is different in a doubly linked list which we will cover in the later chapter.

## Doubly Linked List

After finishing the previous chapters, you should be familiar with the singly linked list.

In this chapter, we are going to introduce another type of linked list: doubly linked list. Different from the singly linked list, the doubly linked list maintains two reference fields in each node.

We will introduce more details in this chapter and help you understand the basic operations in a doubly linked list.

**Introduction - Doubly Linked List**

We have introduced the singly linked list in previous chapters.

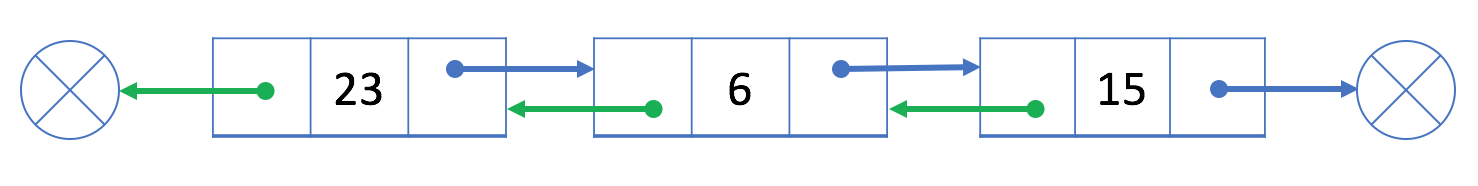
A node in a singly linked list has the value field, and a "next" reference field to link nodes sequentially.

In this article, we will introduce another type of linked list: Doubly Linked List.

### ***Definition***

The doubly linked list works in a similar way but has one more reference field which is known as the "prev" field. With this extra field, you are able to know the previous node of the current node.

Let's take a look at an example:



The green arrows indicate how our "prev" field works.

### ***Node Structure***

Here is a typical definition of the node structure in a doubly linked list:

|  |
| --- |
| // Definition for doubly-linked list.  class DoublyListNode {  int val;  DoublyListNode next, prev;  DoublyListNode(int x) {val = x;}  } |

Similar to the singly linked list, we will use the head node to represent the whole list.

### ***Operations***

Similar to a singly linked list, we will introduce how to access data, insert a new node or delete an existing node in a doubly linked list.

We can access data in the same exact way as in a singly linked list:

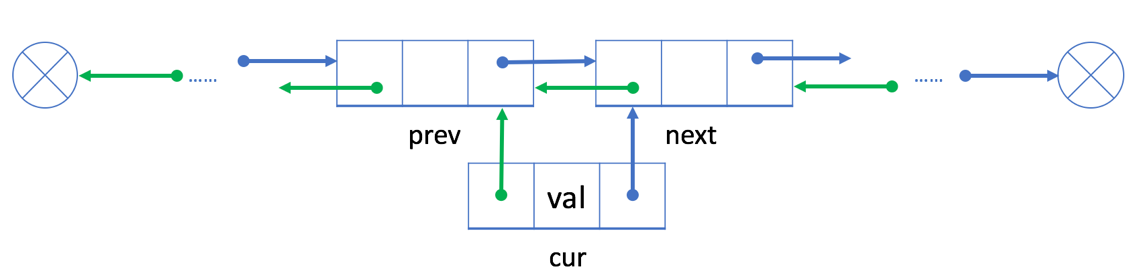
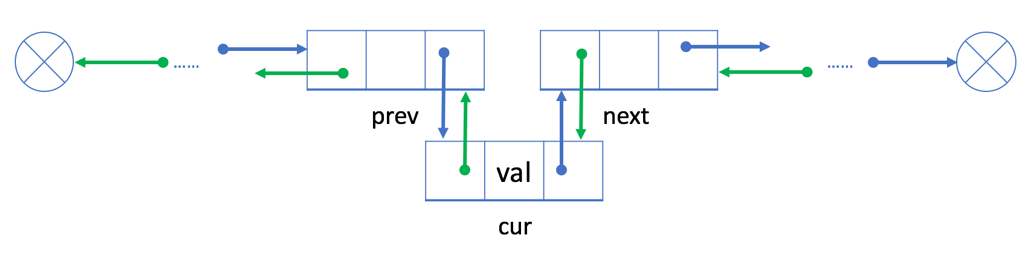
1. We are not able to access a random position in constant time.
2. We have to traverse from the head to get the i-th node we want.
3. The time complexity in the worse case will be O(N), where N is the length of the linked list.

For addition and deletion, it will be a little more complicated since we need to take care of the "prev" field as well. We will go through these two operations in next two articles.

After that, we provide an exercise for you to redesign the linked list using doubly linked list.

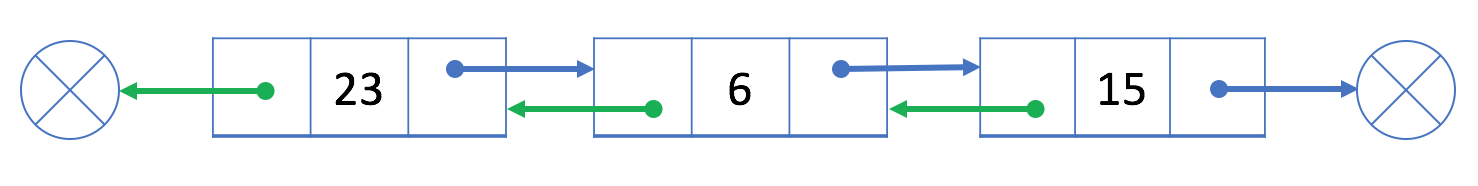
**Add Operation - Doubly Linked List**

If we want to insert a new node cur after an existing node prev, we can divide this process into two steps:

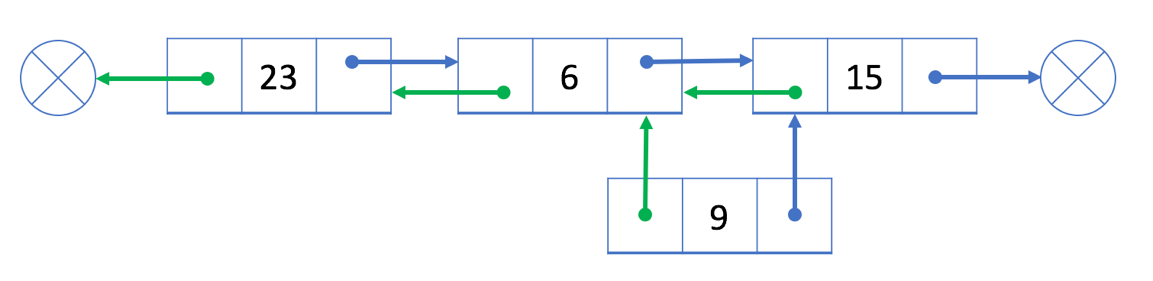
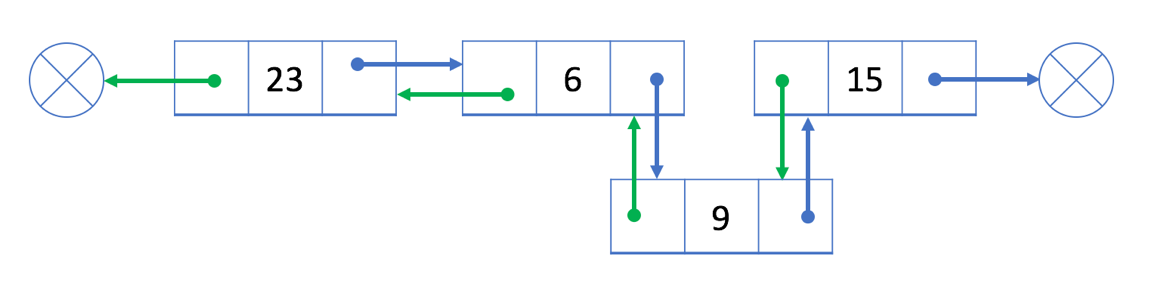
1. link cur with prev and next, where next is the original next node of prev;  
   
2. re-link the prev and next with cur.   
   

Similar to the singly linked list, both the time and the space complexity of the add operation are O(1).

### ***An Example***



Let's add a new node 9 after the existing node 6:

1. link cur (node 9) with prev (node 6) and next (node 15)  
   
2. re-link prev (node 6) and next (node 15) with cur (node 9)  
   

What if we want to insert a new node at the beginning or at the end?

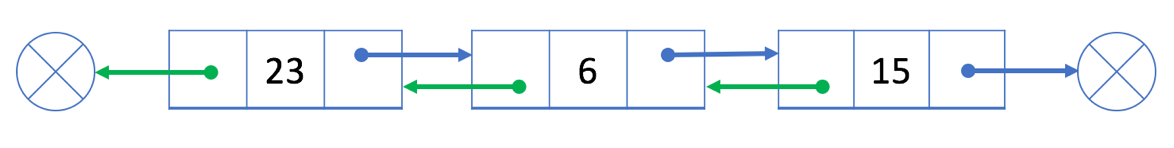
**Delete Operation - Doubly Linked List**

If we want to delete an existing node cur from the doubly linked list, we can simply link its previous node prev with its next node next.

Unlike the singly linked list, it is easy to get the previous node in constant time with the "prev" field.

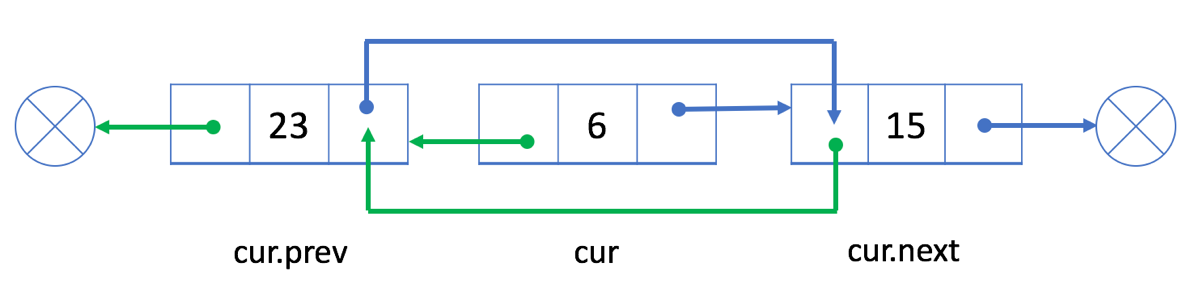
Since we no longer need to traverse the linked list to get the previous node, both the time and space complexity are O(1).

### ***An Example***



Our goal is to delete the node 6 from the doubly linked list.

So we link its previous node 23 and its next node 15:



Node 6 is not in our doubly linked list now.

What if we want to delete the first node or the last node?

**Design Linked List**

Design your implementation of the linked list. You can choose to use a singly or doubly linked list.  
A node in a singly linked list should have two attributes: val and next. val is the value of the current node, and next is a pointer/reference to the next node.  
If you want to use the doubly linked list, you will need one more attribute prev to indicate the previous node in the linked list. Assume all nodes in the linked list are **0-indexed**.

Implement the MyLinkedList class:

* MyLinkedList() Initializes the MyLinkedList object.
* int get(int index) Get the value of the indexth node in the linked list. If the index is invalid, return -1.
* void addAtHead(int val) Add a node of value val before the first element of the linked list. After the insertion, the new node will be the first node of the linked list.
* void addAtTail(int val) Append a node of value val as the last element of the linked list.
* void addAtIndex(int index, int val) Add a node of value val before the indexth node in the linked list. If index equals the length of the linked list, the node will be appended to the end of the linked list. If index is greater than the length, the node **will not be inserted**.
* void deleteAtIndex(int index) Delete the indexth node in the linked list, if the index is valid.

**Example 1:**

**Input**

["MyLinkedList", "addAtHead", "addAtTail", "addAtIndex", "get", "deleteAtIndex", "get"]

[[], [1], [3], [1, 2], [1], [1], [1]]

**Output**

[null, null, null, null, 2, null, 3]

**Explanation**

MyLinkedList myLinkedList = new MyLinkedList();

myLinkedList.addAtHead(1);

myLinkedList.addAtTail(3);

myLinkedList.addAtIndex(1, 2); // linked list becomes 1->2->3

myLinkedList.get(1); // return 2

myLinkedList.deleteAtIndex(1); // now the linked list is 1->3

myLinkedList.get(1); // return 3

**Constraints:**

* 0 <= index, val <= 1000
* Please do not use the built-in LinkedList library.
* At most 2000 calls will be made to get, addAtHead, addAtTail,  addAtIndex and deleteAtIndex.

**Design Doubly Linked List - Solution**

Let's briefly review the structure definition of a node in the doubly linked list.

|  |
| --- |
| // Definition for doubly-linked list.  class DoublyListNode {  int val;  DoublyListNode next, prev;  DoublyListNode(int x) {val = x;}  } |

Based on this definition, we are going to give you the solution step by step. The solution for the doubly linked list is similar to the one using singly linked list.

**1. Initiate a new linked list: represent a linked list using the head node.**

|  |
| --- |
| class MyLinkedList {  private DoublyListNode head;  /\*\* Initialize your data structure here. \*/  public MyLinkedList() {  head = null;  }  } |

**2. Traverse the linked list to get element by index.**

|  |
| --- |
| /\*\* Helper function to return the index-th node in the linked list. \*/  private DoublyListNode getNode(int index) {  DoublyListNode cur = head;  for (int i = 0; i < index && cur != null; ++i) {  cur = cur.next;  }  return cur;  }  /\*\* Helper function to return the last node in the linked list. \*/  private DoublyListNode getTail() {  DoublyListNode cur = head;  while (cur != null && cur.next != null) {  cur = cur.next;  }  return cur;  }  /\*\* Get the value of the index-th node in the linked list. If the index is invalid, return -1. \*/  public int get(int index) {  DoublyListNode cur = getNode(index);  return cur == null ? -1 : cur.val;  } |

**3. Add a new node.**

|  |
| --- |
| /\*\* Add a node of value val before the first element of the linked list. After the insertion, the new node will be the first node of the linked list. \*/  public void addAtHead(int val) {  DoublyListNode cur = new DoublyListNode(val);  cur.next = head;  if (head != null) {  head.prev = cur;  }  head = cur;  return;  }  /\*\* Append a node of value val to the last element of the linked list. \*/  public void addAtTail(int val) {  if (head == null) {  addAtHead(val);  return;  }  DoublyListNode prev = getTail();  DoublyListNode cur = new DoublyListNode(val);  prev.next = cur;  cur.prev = prev;  }  /\*\* Add a node of value val before the index-th node in the linked list. If index equals to the length of linked list, the node will be appended to the end of linked list. If index is greater than the length, the node will not be inserted. \*/  public void addAtIndex(int index, int val) {  if (index == 0) {  addAtHead(val);  return;  }  DoublyListNode prev = getNode(index - 1);  if (prev == null) {  return;  }  DoublyListNode cur = new DoublyListNode(val);  DoublyListNode next = prev.next;  cur.prev = prev;  cur.next = next;  prev.next = cur;  if (next != null) {  next.prev = cur;  }  } |

**5. Delete a node.**

|  |
| --- |
| /\*\* Delete the index-th node in the linked list, if the index is valid. \*/  public void deleteAtIndex(int index) {  DoublyListNode cur = getNode(index);  if (cur == null) {  return;  }  DoublyListNode prev = cur.prev;  DoublyListNode next = cur.next;  if (prev != null) {  prev.next = next;  } else {  // modify head when deleting the first node.  head = next;  }  if (next != null) {  next.prev = prev;  }  } |

Similar to the add operation, it takes O(N) time to get the node by the index which is different from deleting a given node. However, it is different to the singly linked list. When we get the node we want to delete, we don't need to traverse to get its previous node but using the "prev" field instead.

## Conclusion

In previous chapters, we have learned a lot about the singly linked list and doubly linked list.

In this chapter, we will summarize what we have learned and do a short comparison between the linked list and other data structures.

We also provide some exercises for you to practice more about the linked list.

**Summary - Linked List**

### ***Review***

Let's briefly review the performance of the singly linked list and doubly linked list.

They are similar in many operations:

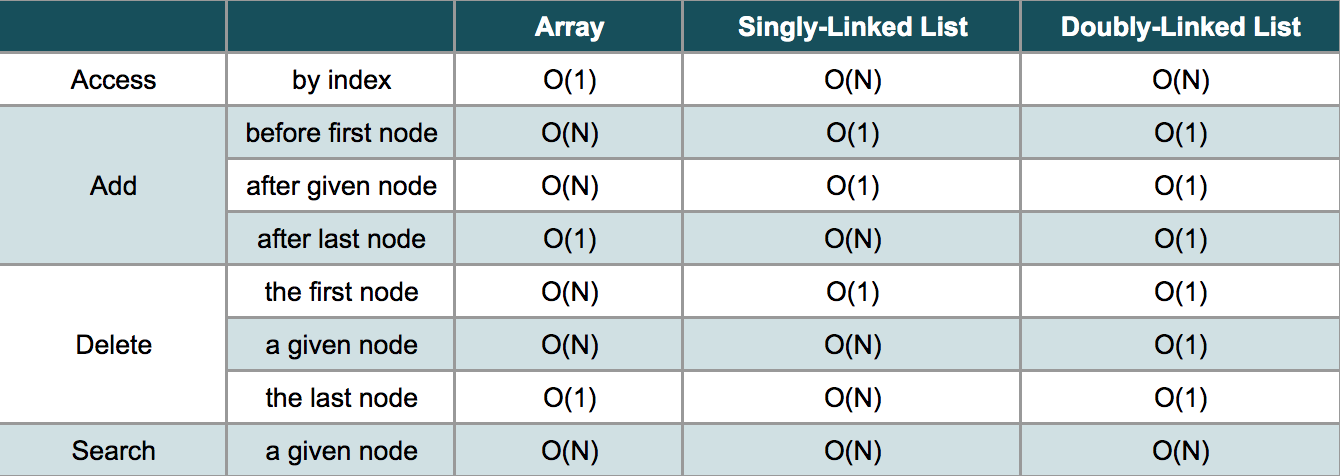
1. Both of them are not able to access the data at a random position in constant time.
2. Both of them are able to add a new node after given node or at the beginning of the list in O(1) time.
3. Both of them are able to delete the first node in O(1) time.

But it is a little different to delete a given node (including the last node).

* In a singly linked list, it is not able to get the previous node of a given node so we have to spend O(N) time to find out the previous node before deleting the given node.
* In a doubly linked list, it will be much easier because we can get the previous node with the "prev" reference field. So we can delete a given node in O(1) time.

### ***Comparison***

Here we provide a comparison of time complexity between the linked list and the [array](https://leetcode.com/explore/learn/card/array-and-string/).



After this comparison, it is not difficult to come up with our conclusion:

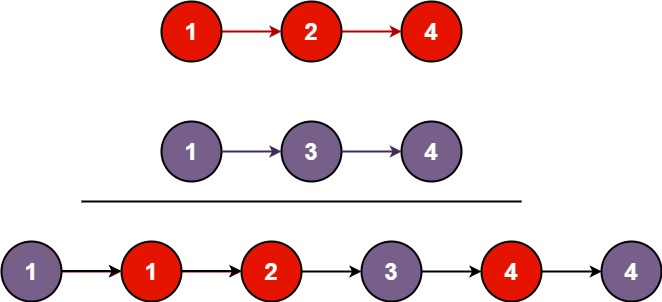
If you need to add or delete a node frequently, a linked list could be a good choice.

If you need to access an element by index often, an array might be a better choice than a linked list.

**Merge Two Sorted Lists**

Merge two sorted linked lists and return it as a **sorted** list. The list should be made by splicing together the nodes of the first two lists.

**Example 1:**



**Input:** l1 = [1,2,4], l2 = [1,3,4]

**Output:** [1,1,2,3,4,4]

**Example 2:**

**Input:** l1 = [], l2 = []

**Output:** []

**Example 3:**

**Input:** l1 = [], l2 = [0]

**Output:** [0]

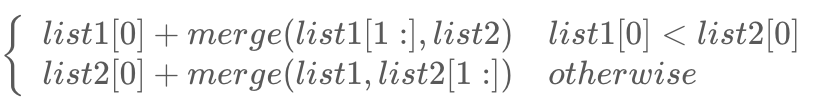
**Constraints:**

* The number of nodes in both lists is in the range [0, 50].
* -100 <= Node.val <= 100
* Both l1 and l2 are sorted in **non-decreasing** order.

#### **Approach 1: Recursion**

**Intuition**

We can recursively define the result of a merge operation on two lists as the following (avoiding the corner case logic surrounding empty lists):



Namely, the smaller of the two lists' heads plus the result of a merge on the rest of the elements.

**Algorithm**

We model the above recurrence directly, first accounting for edge cases. Specifically, if either of l1 or l2 is initially null, there is no merge to perform, so we simply return the non-null list. Otherwise, we determine which of l1 and l2 has a smaller head, and recursively set the next value for that head to the next merge result. Given that both lists are null-terminated, the recursion will eventually terminate.

|  |
| --- |
| class Solution {  public ListNode mergeTwoLists(ListNode l1, ListNode l2) {  if (l1 == null) {  return l2;  }  else if (l2 == null) {  return l1;  }  else if (l1.val < l2.val) {  l1.next = mergeTwoLists(l1.next, l2);  return l1;  }  else {  l2.next = mergeTwoLists(l1, l2.next);  return l2;  }  }  } |

**Complexity Analysis**

* Time complexity : *O*(*n*+*m*)

Because each recursive call increments the pointer to l1 or l2 by one (approaching the dangling null at the end of each list), there will be exactly one call to mergeTwoLists per element in each list. Therefore, the time complexity is linear in the combined size of the lists.

* Space complexity : *O*(*n*+*m*)

The first call to mergeTwoLists does not return until the ends of both l1 and l2 have been reached, so *n*+*m* stack frames consume *O*(*n*+*m*) space.

#### **Approach 2: Iteration**

**Intuition**

We can achieve the same idea via iteration by assuming that l1 is entirely less than l2 and processing the elements one-by-one, inserting elements of l2 in the necessary places in l1.

**Algorithm**

First, we set up a false "prehead" node that allows us to easily return the head of the merged list later. We also maintain a prev pointer, which points to the current node for which we are considering adjusting its next pointer. Then, we do the following until at least one of l1 and l2 points to null: if the value at l1 is less than or equal to the value at l2, then we connect l1 to the previous node and increment l1. Otherwise, we do the same, but for l2. Then, regardless of which list we connected, we increment prev to keep it one step behind one of our list heads.

After the loop terminates, at most one of l1 and l2 is non-null. Therefore (because the input lists were in sorted order), if either list is non-null, it contains only elements greater than all of the previously-merged elements. This means that we can simply connect the non-null list to the merged list and return it.

To see this in action on an example, check out the animation below:

|  |
| --- |
| class Solution {  public ListNode mergeTwoLists(ListNode l1, ListNode l2) {  // maintain an unchanging reference to node ahead of the return node.  ListNode prehead = new ListNode(-1);  ListNode prev = prehead;  while (l1 != null && l2 != null) {  if (l1.val <= l2.val) {  prev.next = l1;  l1 = l1.next;  } else {  prev.next = l2;  l2 = l2.next;  }  prev = prev.next;  }  // exactly one of l1 and l2 can be non-null at this point, so connect  // the non-null list to the end of the merged list.  prev.next = l1 == null ? l2 : l1;  return prehead.next;  }  } |

**Complexity Analysis**

* Time complexity : *O*(*n*+*m*)

Because exactly one of l1 and l2 is incremented on each loop iteration, the while loop runs for a number of iterations equal to the sum of the lengths of the two lists. All other work is constant, so the overall complexity is linear.

* Space complexity : *O*(1)

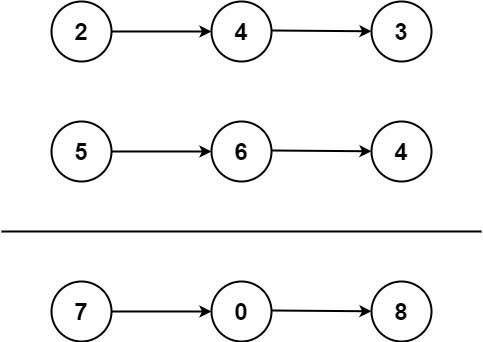
The iterative approach only allocates a few pointers, so it has a constant overall memory footprint.

**Add Two Numbers**

You are given two **non-empty** linked lists representing two non-negative integers. The digits are stored in **reverse order**, and each of their nodes contains a single digit. Add the two numbers and return the sum as a linked list.

You may assume the two numbers do not contain any leading zero, except the number 0 itself.

**Example 1:**



**Input:** l1 = [2,4,3], l2 = [5,6,4]

**Output:** [7,0,8]

**Explanation:** 342 + 465 = 807.

**Example 2:**

**Input:** l1 = [0], l2 = [0]

**Output:** [0]

**Example 3:**

**Input:** l1 = [9,9,9,9,9,9,9], l2 = [9,9,9,9]

**Output:** [8,9,9,9,0,0,0,1]

**Constraints:**

* The number of nodes in each linked list is in the range [1, 100].
* 0 <= Node.val <= 9
* It is guaranteed that the list represents a number that does not have leading zeros.

## Solution

#### **Approach 1: Elementary Math**

**Intuition**

Keep track of the carry using a variable and simulate digits-by-digits sum starting from the head of list, which contains the least-significant digit.

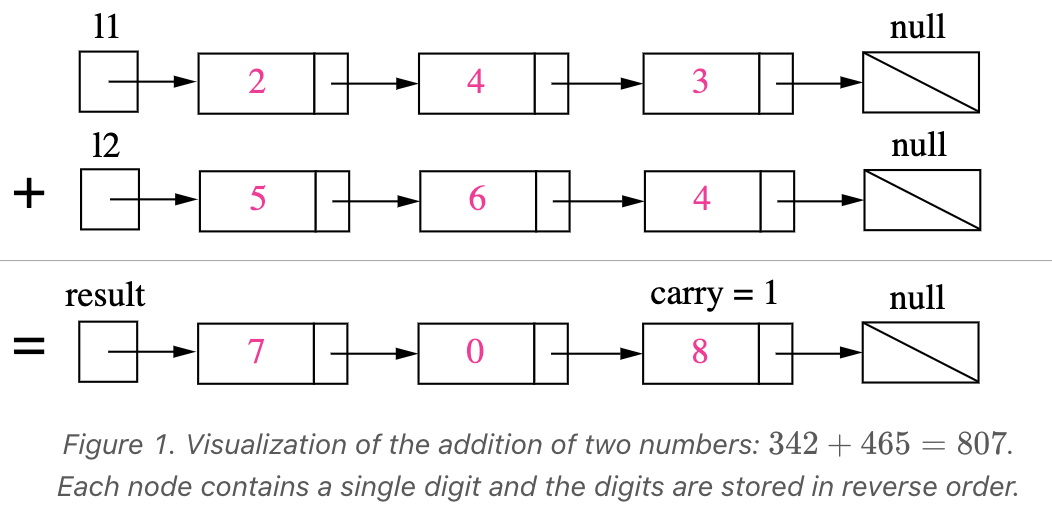


Figure 1. Visualization of the addition of two numbers: *342 + 465 = 807342+465=807*.Each node contains a single digit and the digits are stored in reverse order.

**Algorithm**

Just like how you would sum two numbers on a piece of paper, we begin by summing the least-significant digits, which is the head of *l*1 and *l*2. Since each digit is in the range of 0…9, summing two digits may "overflow". For example 5+7=12. In this case, we set the current digit to 2 and bring over the *carry*=1 to the next iteration. *carry* must be either 0 or 1 because the largest possible sum of two digits (including the carry) is 9+9+1=19.

The pseudocode is as following:

* Initialize current node to dummy head of the returning list.
* Initialize carry to 0.
* Initialize p*p* and *q* to head of l1*l*1 and *l*2 respectively.
* Loop through lists l1*l*1 and *l*2 until you reach both ends.
  + Set *x* to node p*p*'s value. If *p* has reached the end of *l*1, set to 0.
  + Set *y* to node q*q*'s value. If *q* has reached the end of *l*2, set to 0.
  + Set *sum*=*x*+*y*+*carry*.
  + Update *carry*=*sum*/10.
  + Create a new node with the digit value of (*sum* mod 10) and set it to current node's next, then advance current node to next.
  + Advance both *p* and *q*.
* Check if *carry*=1, if so append a new node with digit 1 to the returning list.
* Return dummy head's next node.

Note that we use a dummy head to simplify the code. Without a dummy head, you would have to write extra conditional statements to initialize the head's value.

Take extra caution of the following cases:

| **Test case** | **Explanation** |
| --- | --- |
| *l*1=[0,1] *l*2=[0,1,2] | When one list is longer than the other. |
| *l*1=[] *l*2=[0,1] | When one list is null, which means an empty list. |
| *l*1=[9,9] *l*2=[1] | The sum could have an extra carry of one at the end, which is easy to forget. |

|  |
| --- |
| public ListNode addTwoNumbers(ListNode l1, ListNode l2) {  ListNode dummyHead = new ListNode(0);  ListNode p = l1, q = l2, curr = dummyHead;  int carry = 0;  while (p != null || q != null) {  int x = (p != null) ? p.val : 0;  int y = (q != null) ? q.val : 0;  int sum = carry + x + y;  carry = sum / 10;  curr.next = new ListNode(sum % 10);  curr = curr.next;  if (p != null) p = p.next;  if (q != null) q = q.next;  }  if (carry > 0) {  curr.next = new ListNode(carry);  }  return dummyHead.next;  } |

**Complexity Analysis**

* Time complexity : *O*(max(*m*,*n*)). Assume that m*m* and *n* represents the length of *l*1 and *l*2 respectively, the algorithm above iterates at most max(*m*,*n*) times.
* Space complexity : *O*(max(*m*,*n*)). The length of the new list is at most max(*m*,*n*)+1.

**Follow up**

What if the the digits in the linked list are stored in non-reversed order? For example:

(3→4→2)+(4→6→5)=8→0→7

**Flatten a Multilevel Doubly Linked List**

You are given a doubly linked list which in addition to the next and previous pointers, it could have a child pointer, which may or may not point to a separate doubly linked list. These child lists may have one or more children of their own, and so on, to produce a multilevel data structure, as shown in the example below.

Flatten the list so that all the nodes appear in a single-level, doubly linked list. You are given the head of the first level of the list.

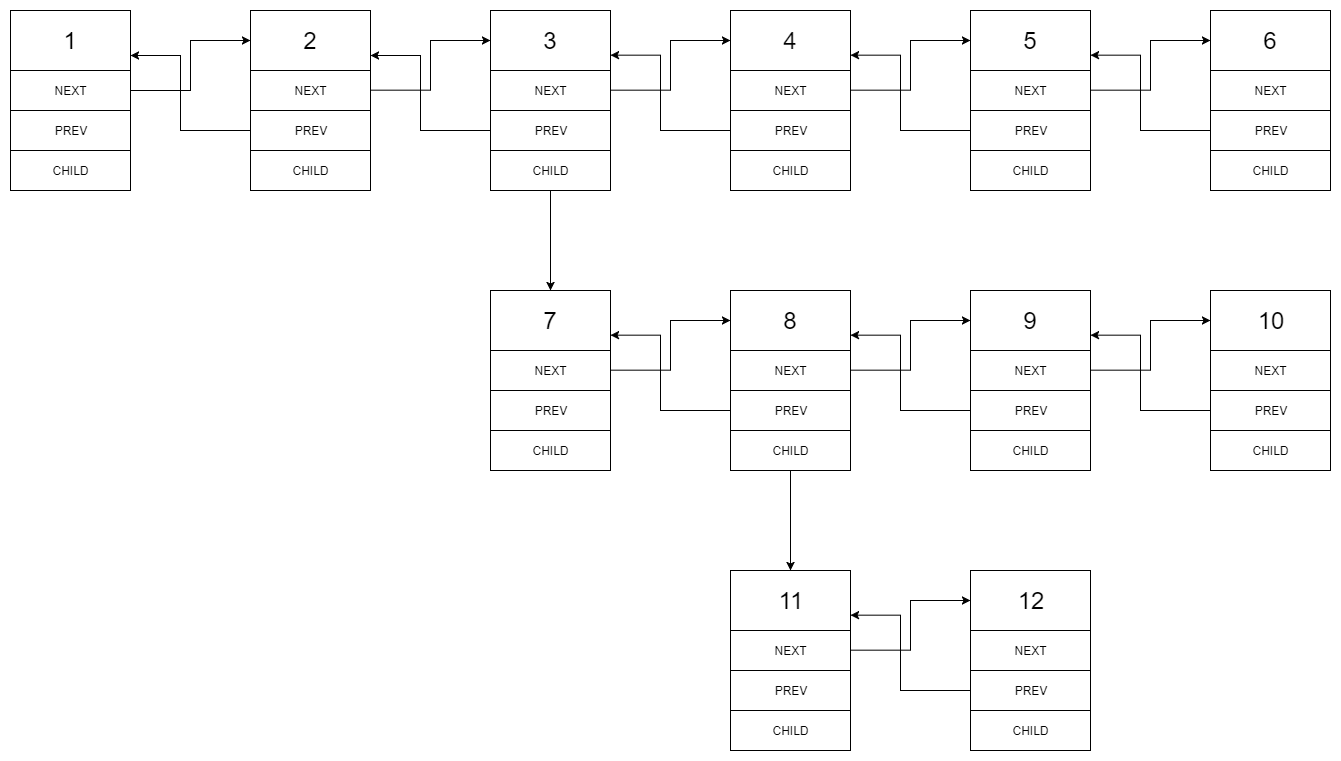
**Example 1:**

**Input:** head = [1,2,3,4,5,6,null,null,null,7,8,9,10,null,null,11,12]

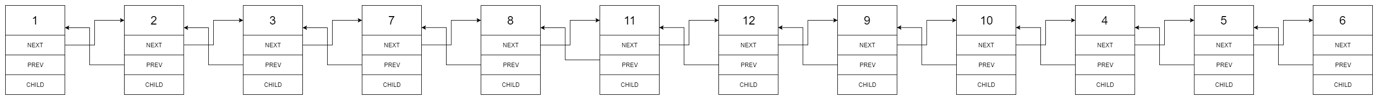
**Output:** [1,2,3,7,8,11,12,9,10,4,5,6]

**Explanation:**

The multilevel linked list in the input is as follows:



After flattening the multilevel linked list it becomes:



**Example 2:**

**Input:** head = [1,2,null,3]

**Output:** [1,3,2]

**Explanation:**

The input multilevel linked list is as follows:

1---2---NULL

|

3---NULL

**Example 3:**

**Input:** head = []

**Output:** []

**How multilevel linked list is represented in test case:**

We use the multilevel linked list from **Example 1** above:

1---2---3---4---5---6--NULL

|

7---8---9---10--NULL

|

11--12--NULL

The serialization of each level is as follows:

[1,2,3,4,5,6,null]

[7,8,9,10,null]

[11,12,null]

To serialize all levels together we will add nulls in each level to signify no node connects to the upper node of the previous level. The serialization becomes:

[1,2,3,4,5,6,null]

[null,null,7,8,9,10,null]

[null,11,12,null]

Merging the serialization of each level and removing trailing nulls we obtain:

[1,2,3,4,5,6,null,null,null,7,8,9,10,null,null,11,12]

**Constraints:**

* The number of Nodes will not exceed 1000.
* 1 <= Node.val <= 105

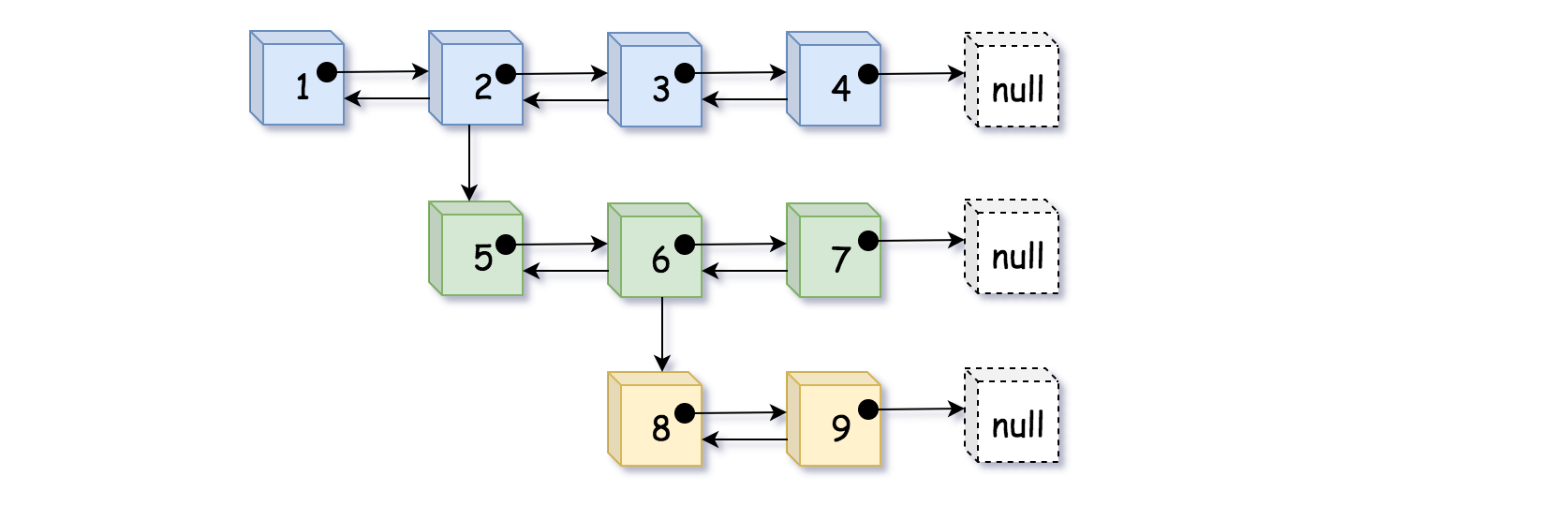
## Solution

#### **Approach 1: DFS by Recursion**

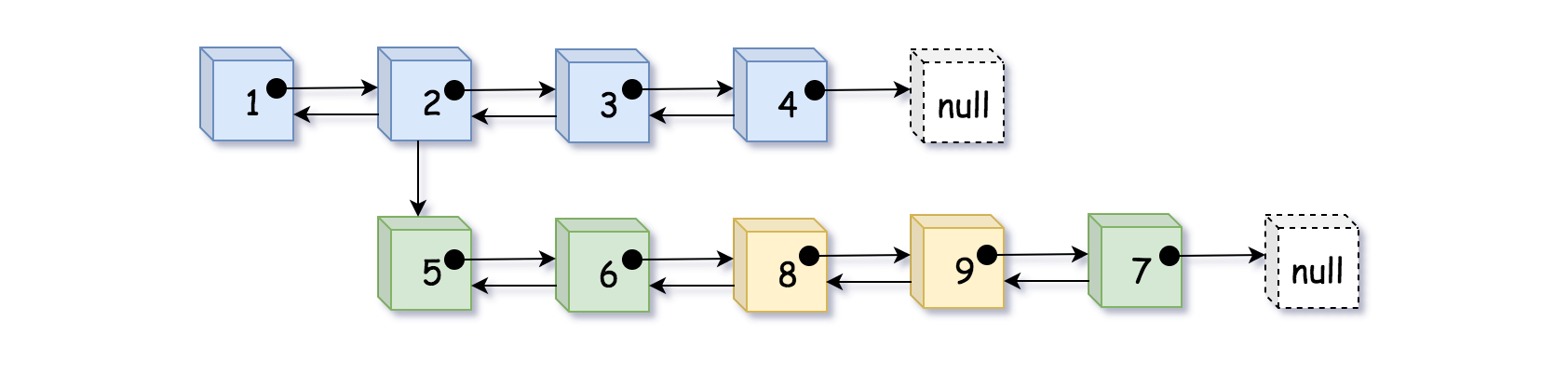
**Intuition**

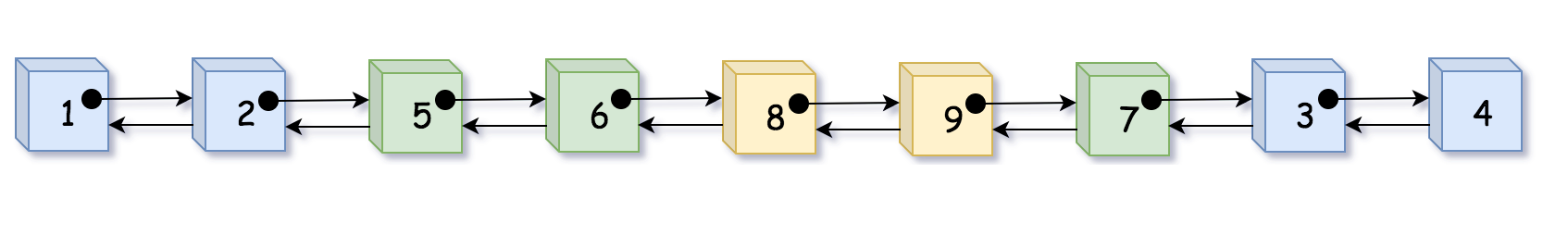
People might ask themselves in which scenario that one would use such an awkward data structure. Well, one of the scenarios could be a simplified version of git branching. By flattening the multilevel list, one can think it as merging all git branches together, though it is not at all how the git merge works.

First of all, to clarify what is the desired result of the flatten operation, we illustrate with an example below.



In the above example, we distinguish nodes in different levels with different colors. We could flatten the list in two steps as follows:

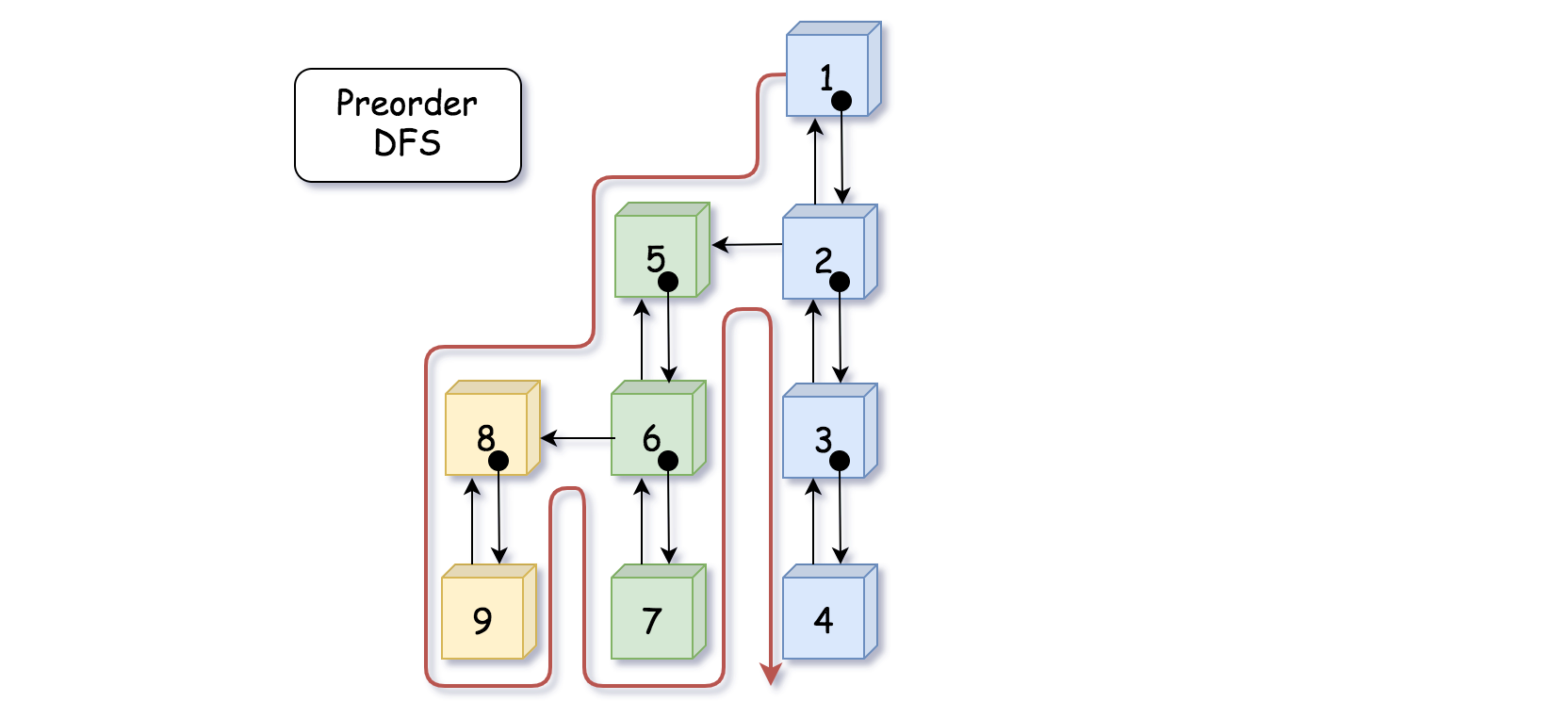




As we can see, by flatten, we basically **fold or embed** the sublist that is branched from the child pointer into its parent list.

This is one way to interpret the flatten operation. However, as intuitive as the problem seems to be, one might stumble over the implementation. It is because the above intuition does not quite catch the true nature of the problem.

Actually, if we turn the above list in 90 degrees around the clock, then suddenly a **binary tree** appear in front of us. And the flatten operation is basically what we call ***preorder DFS traversal*** (Depth-First Search).



Indeed, as shown in the above graph, we could consider the child pointer as the left pointer in binary tree which points to the left sub-tree (sublist). And similarly, the next pointer can be considered as the right pointer in binary tree. Then if we traverse the tree in preorder DFS, it would generate the same visiting sequence as the flatten operation in our problem.

**Algorithm**

Now that we know the problem is basically asking us to do a preorder DFS traversal over a disguised binary tree, we could use this intuition to guide the implementation.

As many of you would know that there are generally two manners to implement the DFS traversal: recursion and iteration. We here start with the recursion, since many find it more intuitive.

Here it goes with the recursive DFS algorithm:

* First of all, we define our recursive function as flatten\_dfs(prev, curr) which takes two pointers as input and returns the pointer of tail in the flattened list. The curr pointer leads to the sub-list that we would like to flatten, and the prev pointer points to the element that should precede the curr element.
* Within the recursive function flatten\_dfs(prev, curr), we first establish the double links between the prev and curr nodes, as in the ***preorder*** DFS we take care of the **current state** first before looking into the children.
* Further in the function flatten\_dfs(prev, curr), we then go ahead to flatten the **left subtree** (i.e. the sublist pointed by the curr.child pointer) with the call of flatten\_dfs(curr, curr.child), which returns the tail element to the flattened sublist. Then with the tail element of the previous sublist, we then further flatten the **right subtree** (i.e. the sublist pointed by the curr.next pointer), with the call of flatten\_dfs(tail, curr.next).
* And voila, that is our core function. There are two additional important details that we should attend to, in order to obtain the correct result:
  + We should make a copy of the curr.next pointer before the first recursive call of flatten\_dfs(curr, curr.child), since the curr.next pointer might be altered within the function.
  + After we flatten the sublist pointed by the curr.child pointer, we should remove the child pointer since it is no longer needed in the final result.
* Last by not the least, one would notice in the following implementation that we create a pseudoHead variable in the function. This is not absolutely necessary, but it would help us to make the solution more concise and elegant by **reducing the null pointer checks** (e.g. if prev == null). And with less branching tests, it certainly helps with the performance as well. Sometimes people might call it ***sentinel*** node. As one might have seen before, this is a useful trick that one could apply to many problems related with linked lists (e.g. [LRU cache](https://leetcode.com/articles/lru-cache/)).

|  |
| --- |
| /\*  // Definition for a Node.  class Node {  public int val;  public Node prev;  public Node next;  public Node child;  public Node() {}  public Node(int \_val,Node \_prev,Node \_next,Node \_child) {  val = \_val;  prev = \_prev;  next = \_next;  child = \_child;  }  };  \*/  class Solution {  public Node flatten(Node head) {  if (head == null) return head;  // pseudo head to ensure the `prev` pointer is never none  Node pseudoHead = new Node(0, null, head, null);  flattenDFS(pseudoHead, head);  // detach the pseudo head from the real head  pseudoHead.next.prev = null;  return pseudoHead.next;  }  /\* return the tail of the flatten list \*/  public Node flattenDFS(Node prev, Node curr) {  if (curr == null) return prev;  curr.prev = prev;  prev.next = curr;  // the curr.next would be tempered in the recursive function  Node tempNext = curr.next;  Node tail = flattenDFS(curr, curr.child);  curr.child = null;  return flattenDFS(tail, tempNext);  }  } |

**Complexity**

* Time Complexity: O(*N*) where N*N* is the number of nodes in the list. The DFS algorithm traverses each node once and only once.
* Space Complexity: O(*N*) where N*N* is the number of nodes in the list. In the worst case, the binary tree might be extremely unbalanced (i.e. the tree leans to the left), which corresponds to the case where nodes are chained with each other only with the child pointers. In this case, the recursive calls would pile up, and it would take N*N* space in the function call stack.

#### **Approach 2: DFS by Iteration**

**Intuition**

Following the intuition of the above DFS preorder traversal approach, here we demonstrate how one can implement the solution via **iteration**.

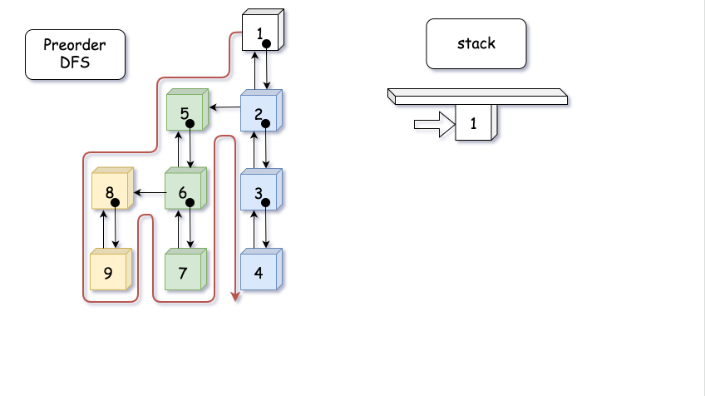
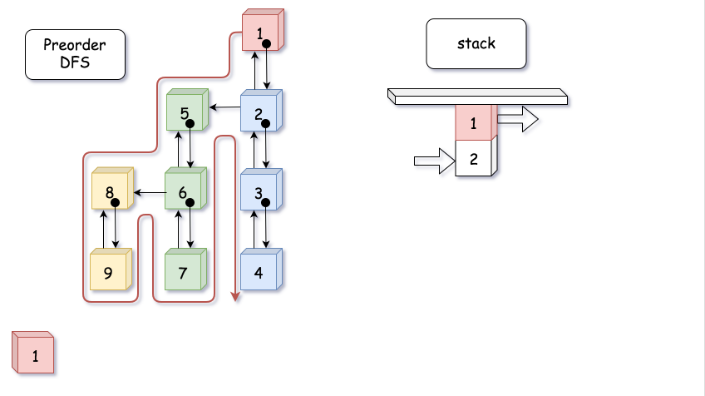
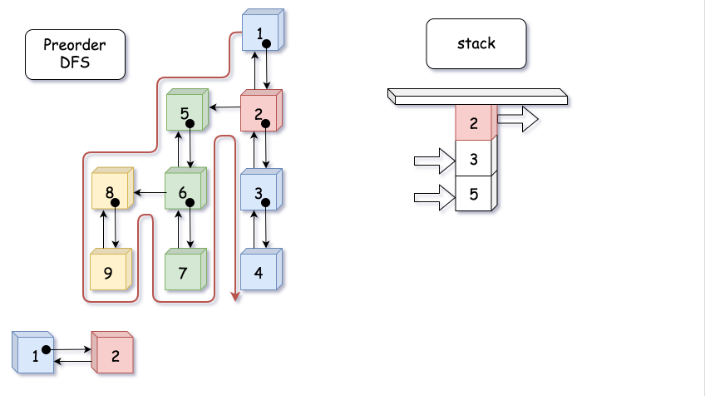
The key is to use the data structure called ***stack***, which is a container that follows the principle of LIFO (last in, first out). The element that enters the stack at last would come out first, similar with the scenario of a packed elevator.

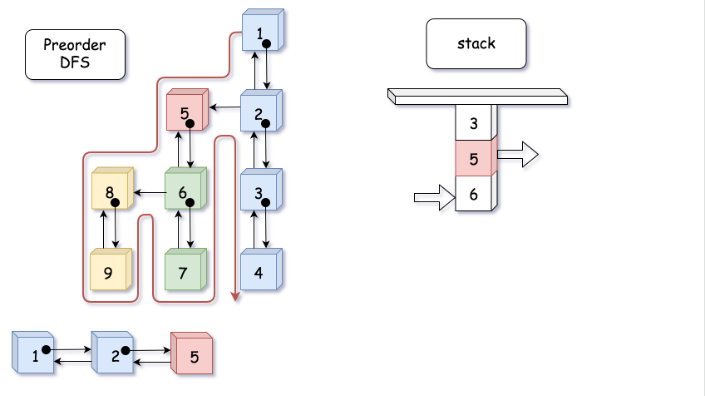
The stack data structure helps us to construct the iteration sequence as the one created by recursion. The stack here mimics the behavior of the function call stack, so that we could obtain the same result without resorting to recursion.

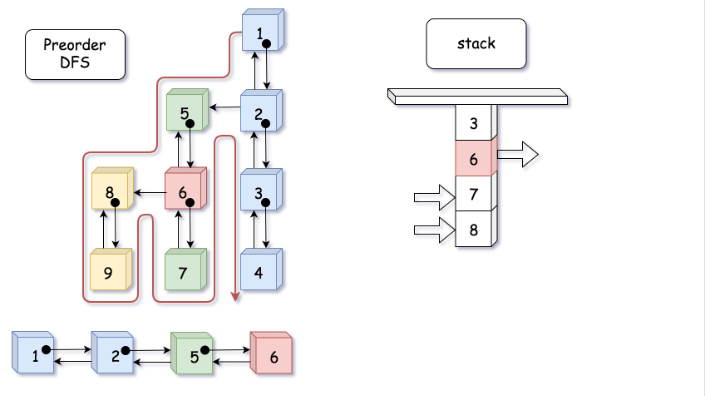
**Algorithm**

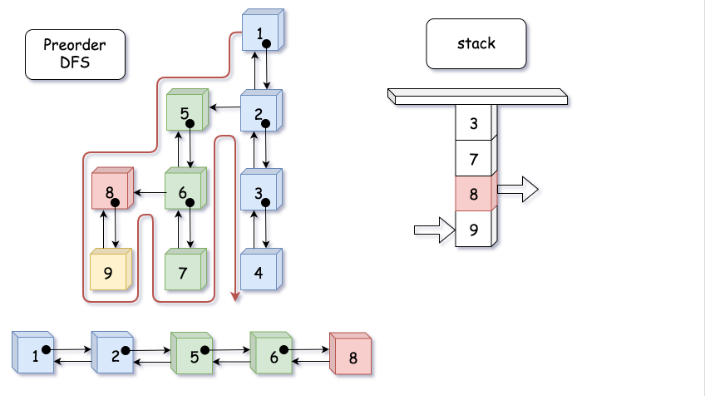
* First of all, we create a stack and then we push the head node to the stack. In addition, we create a variable called prev which would help us to track the precedent node at each step during the iteration.
* Then we enter a loop to iterate the stack until the stack becomes empty.
* Within the loop, at each step, we first pop out a node (named curr) from the stack. Then we establish the links between prev and curr. Then in the following steps, we take care of the nodes pointed by the curr.next and curr.child pointers respectively, and strictly in this order.
  + First, if the curr.next does exist (i.e. there exists a right subtree), we then push the node into the stack for the next iteration.
  + Secondly, if the curr.child does exist (i.e. there exists a left subtree), we then push the node into the stack. In addition, unlike the child.next pointer, we need to clean up the curr.child pointer since it should not be present in the final result.
* And voila. Lastly, we also employ the pseudoHead node to render the algorithm more elegant, as we discussed in the previous approach.

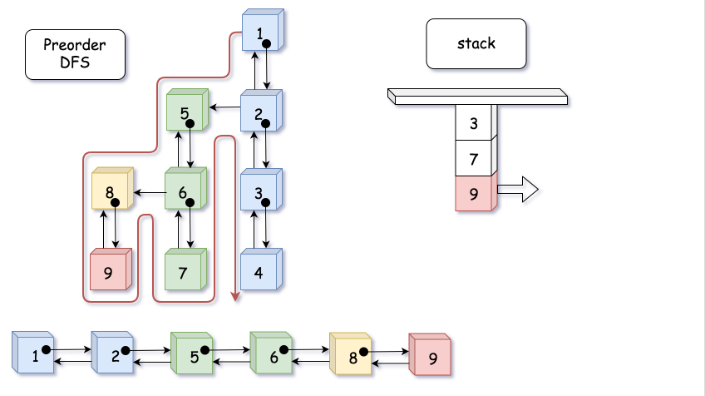
To better illustrate how the algorithm works, we create an animation that shows the evolution of stack step by step, as follows:

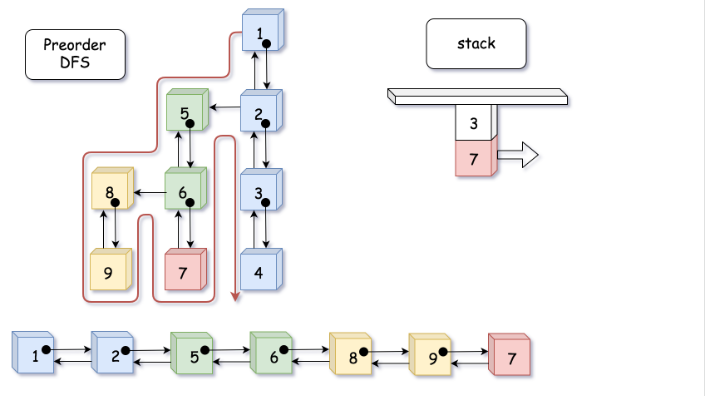
  
  


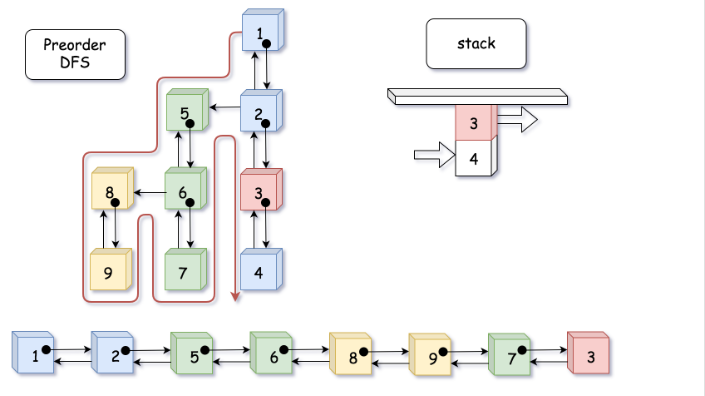


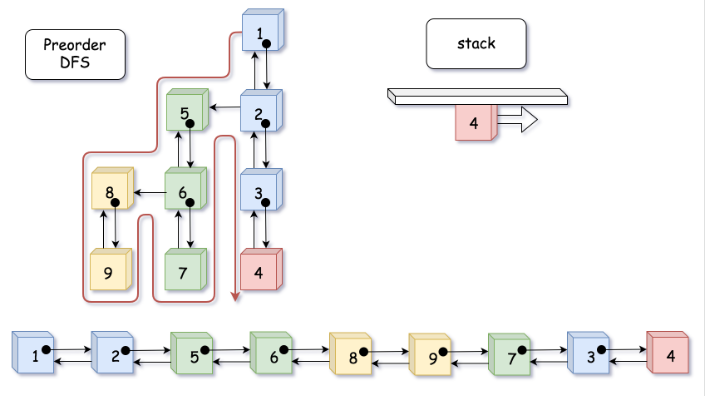


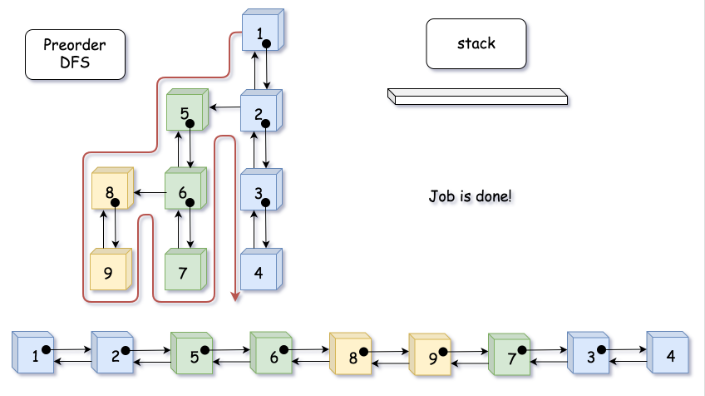












Here are some sample implementations.

|  |
| --- |
| /\*  // Definition for a Node.  class Node {  public int val;  public Node prev;  public Node next;  public Node child;  public Node() {}  public Node(int \_val,Node \_prev,Node \_next,Node \_child) {  val = \_val;  prev = \_prev;  next = \_next;  child = \_child;  }  };  \*/  class Solution {  public Node flatten(Node head) {  if (head == null) return head;  Node pseudoHead = new Node(0, null, head, null);  Node curr, prev = pseudoHead;  Deque<Node> stack = new ArrayDeque<>();  stack.push(head);  while (!stack.isEmpty()) {  curr = stack.pop();  prev.next = curr;  curr.prev = prev;  if (curr.next != null) stack.push(curr.next);  if (curr.child != null) {  stack.push(curr.child);  // don't forget to remove all child pointers.  curr.child = null;  }  prev = curr;  }  // detach the pseudo node from the result  pseudoHead.next.prev = null;  return pseudoHead.next;  }  } |

**Complexity**

* Time Complexity: O(*N*). The iterative solution has the same time complexity as the recursive.
* Space Complexity: O(*N*). Again, the iterative solution has the same space complexity as the recursive one.

**LRU Cache**

Design a data structure that follows the constraints of a [**Least Recently Used (LRU) cache**](https://en.wikipedia.org/wiki/Cache_replacement_policies#LRU).

Implement the LRUCache class:

* LRUCache(int capacity) Initialize the LRU cache with **positive** size capacity.
* int get(int key) Return the value of the key if the key exists, otherwise return -1.
* void put(int key, int value) Update the value of the key if the key exists. Otherwise, add the key-value pair to the cache. If the number of keys exceeds the capacity from this operation, **evict** the least recently used key.

**Follow up:**  
Could you do get and put in O(1) time complexity?

**Example 1:**

**Input**

["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]

[[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]

**Output**

[null, null, null, 1, null, -1, null, -1, 3, 4]

**Explanation**

LRUCache lRUCache = new LRUCache(2);

lRUCache.put(1, 1); // cache is {1=1}

lRUCache.put(2, 2); // cache is {1=1, 2=2}

lRUCache.get(1); // return 1

lRUCache.put(3, 3); // LRU key was 2, evicts key 2, cache is {1=1, 3=3}

lRUCache.get(2); // returns -1 (not found)

lRUCache.put(4, 4); // LRU key was 1, evicts key 1, cache is {4=4, 3=3}

lRUCache.get(1); // return -1 (not found)

lRUCache.get(3); // return 3

lRUCache.get(4); // return 4

**Constraints:**

* 1 <= capacity <= 3000
* 0 <= key <= 3000
* 0 <= value <= 104
* At most 3 \* 104 calls will be made to get and put.

## Solution

#### **Approach 1: Ordered dictionary**

**Intuition**

We're asked to implement [the structure](https://en.wikipedia.org/wiki/Cache_replacement_policies#LRU) which provides the following operations in O(1) time :

* Get the key / Check if the key exists
* Put the key
* Delete the first added key

The first two operations in O(1) time are provided by the standard hashmap, and the last one - by linked list.

There is a structure called ordered dictionary, it combines behind both hashmap and linked list. In Python this structure is called [OrderedDict](https://docs.python.org/3/library/collections.html" \l "collections.OrderedDict) and in Java [LinkedHashMap](https://docs.oracle.com/javase/8/docs/api/java/util/LinkedHashMap.html).

Let's use this structure here.

**Implementation**

|  |
| --- |
| class LRUCache extends LinkedHashMap<Integer, Integer>{  private int capacity;    public LRUCache(int capacity) {  super(capacity, 0.75F, true);  this.capacity = capacity;  }  public int get(int key) {  return super.getOrDefault(key, -1);  }  public void put(int key, int value) {  super.put(key, value);  }  @Override  protected boolean removeEldestEntry(Map.Entry<Integer, Integer> eldest) {  return size() > capacity;  }  }  /\*\*  \* Your LRUCache object will be instantiated and called as such:  \* LRUCache obj = new LRUCache(capacity);  \* int param\_1 = obj.get(key);  \* obj.put(key,value);  \*/ |

**Complexity Analysis**

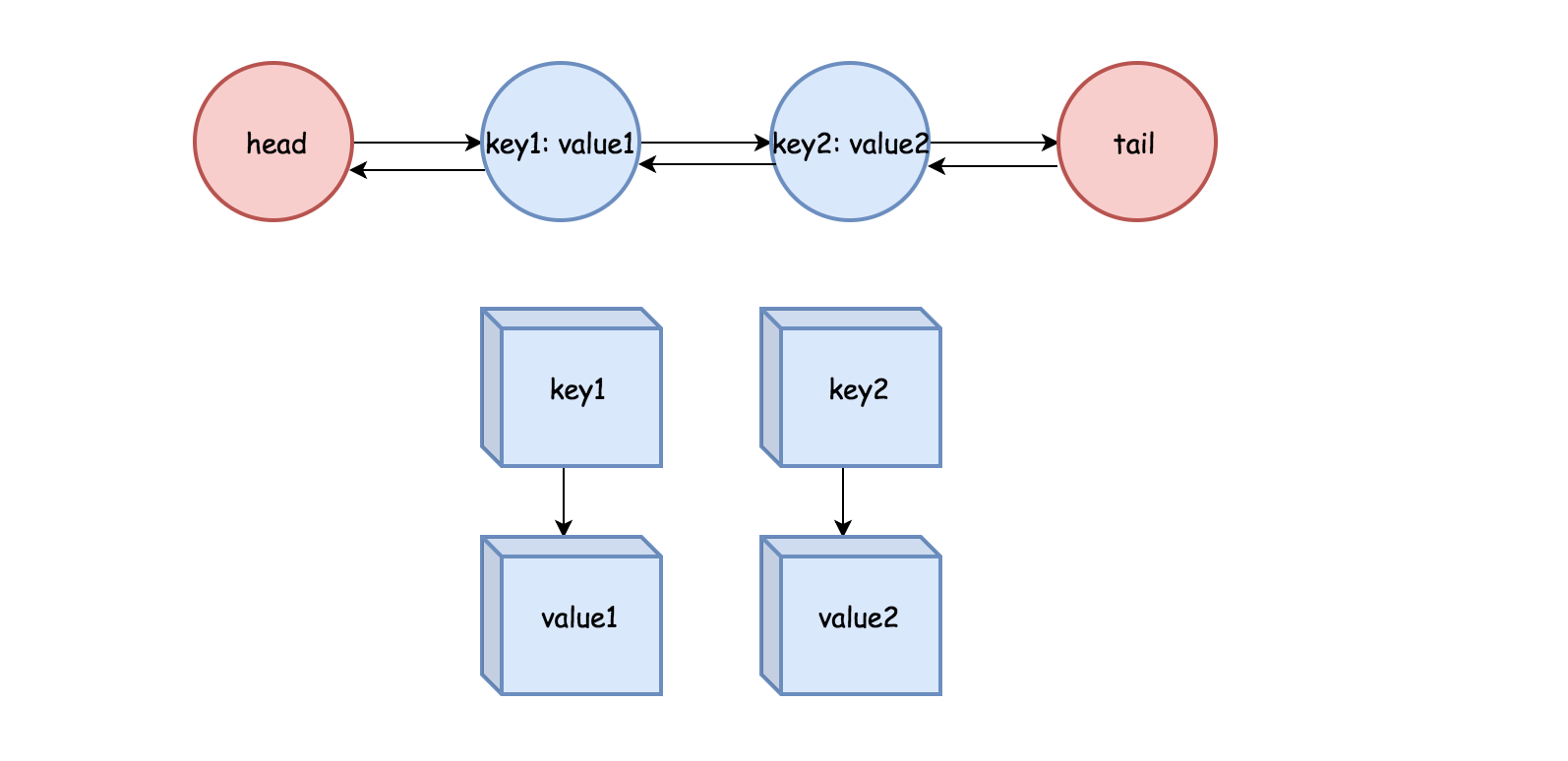
* Time complexity : O(1) both for put and get since all operations with ordered dictionary : get/in/set/move\_to\_end/popitem (get/containsKey/put/remove) are done in a constant time.
* Space complexity : O(*capacity*) since the space is used only for an ordered dictionary with at most capacity + 1 elements.

#### **Approach 2: Hashmap + DoubleLinkedList**

**Intuition**

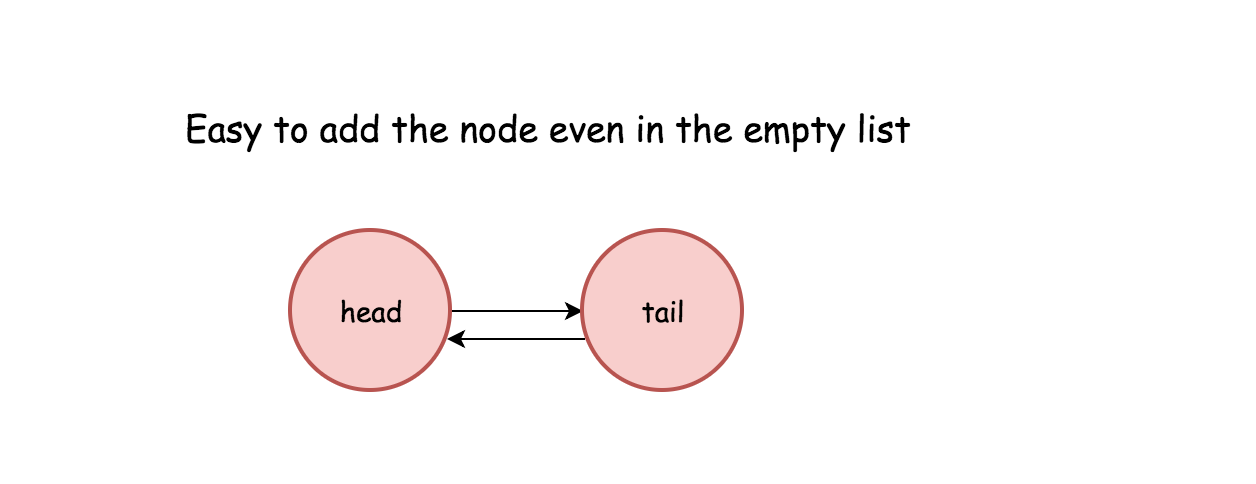
This Java solution is an extended version of the [the article published on the Discuss forum](https://leetcode.com/problems/lru-cache/discuss/45911/Java-Hashtable-%2B-Double-linked-list-(with-a-touch-of-pseudo-nodes)).

The problem can be solved with a hashmap that keeps track of the keys and its values in the double linked list. That results in O(1) time for put and get operations and allows to remove the first added node in O(1) time as well.



One advantage of double linked list is that the node can remove itself without other reference. In addition, it takes constant time to add and remove nodes from the head or tail.

One particularity about the double linked list implemented here is that there are pseudo head and pseudo tail to mark the boundary, so that we don't need to check the null node during the update.



**Implementation**

|  |
| --- |
| public class LRUCache {  class DLinkedNode {  int key;  int value;  DLinkedNode prev;  DLinkedNode next;  }  private void addNode(DLinkedNode node) {  /\*\*  \* Always add the new node right after head.  \*/  node.prev = head;  node.next = head.next;  head.next.prev = node;  head.next = node;  }  private void removeNode(DLinkedNode node){  /\*\*  \* Remove an existing node from the linked list.  \*/  DLinkedNode prev = node.prev;  DLinkedNode next = node.next;  prev.next = next;  next.prev = prev;  }  private void moveToHead(DLinkedNode node){  /\*\*  \* Move certain node in between to the head.  \*/  removeNode(node);  addNode(node);  }  private DLinkedNode popTail() {  /\*\*  \* Pop the current tail.  \*/  DLinkedNode res = tail.prev;  removeNode(res);  return res;  }  private Map<Integer, DLinkedNode> cache = new HashMap<>();  private int size;  private int capacity;  private DLinkedNode head, tail;  public LRUCache(int capacity) {  this.size = 0;  this.capacity = capacity;  head = new DLinkedNode();  // head.prev = null;  tail = new DLinkedNode();  // tail.next = null;  head.next = tail;  tail.prev = head;  }  public int get(int key) {  DLinkedNode node = cache.get(key);  if (node == null) return -1;  // move the accessed node to the head;  moveToHead(node);  return node.value;  }  public void put(int key, int value) {  DLinkedNode node = cache.get(key);  if(node == null) {  DLinkedNode newNode = new DLinkedNode();  newNode.key = key;  newNode.value = value;  cache.put(key, newNode);  addNode(newNode);  ++size;  if(size > capacity) {  // pop the tail  DLinkedNode tail = popTail();  cache.remove(tail.key);  --size;  }  } else {  // update the value.  node.value = value;  moveToHead(node);  }  }  } |

**Complexity Analysis**

* Time complexity : O(1) both for put and get.
* Space complexity : O(*capacity*) since the space is used only for a hashmap and double linked list with at most capacity + 1 elements.

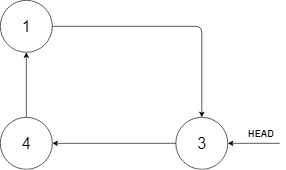
**Insert into a Cyclic Sorted List**

Given a node from a **Circular Linked List** which is sorted in ascending order, write a function to insert a value insertVal into the list such that it remains a sorted circular list. The given node can be a reference to any single node in the list, and may not be necessarily the smallest value in the circular list.

If there are multiple suitable places for insertion, you may choose any place to insert the new value. After the insertion, the circular list should remain sorted.

If the list is empty (i.e., given node is null), you should create a new single circular list and return the reference to that single node. Otherwise, you should return the original given node.

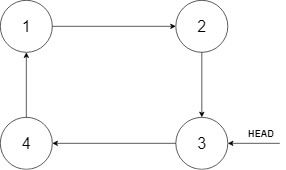
**Example 1:**



**Input:** head = [3,4,1], insertVal = 2

**Output:** [3,4,1,2]

**Explanation:** In the figure above, there is a sorted circular list of three elements. You are given a reference to the node with value 3, and we need to insert 2 into the list. The new node should be inserted between node 1 and node 3. After the insertion, the list should look like this, and we should still return node 3.



**Example 2:**

**Input:** head = [], insertVal = 1

**Output:** [1]

**Explanation:** The list is empty (given head is null). We create a new single circular list and return the reference to that single node.

**Example 3:**

**Input:** head = [1], insertVal = 0

**Output:** [1,0]

**Constraints:**

* 0 <= Number of Nodes <= 5 \* 10^4
* -10^6 <= Node.val <= 10^6
* -10^6 <= insertVal <= 10^6

## Solution

#### **Approach 1: Two-Pointers Iteration**

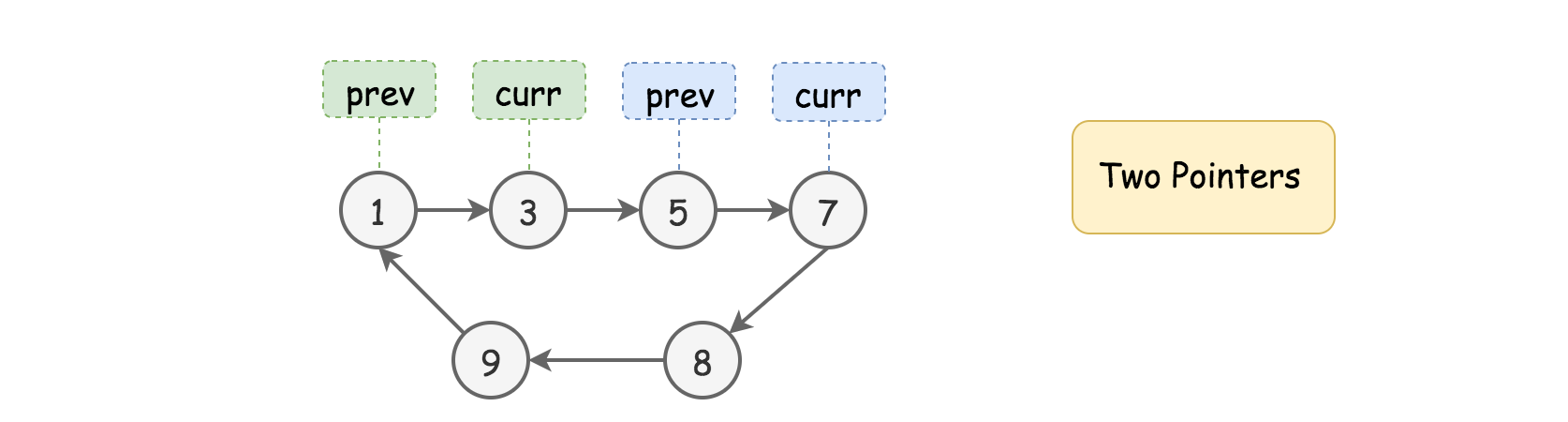
**Intuition**

As simple as the problem might seem to be, it is actually not trivial to write a solution that covers all cases.

Often the case for the problems with linked list, one could apply the approach of **Two-Pointers Iteration**, where one uses two pointers as surrogate to traverse the linked list.

One of reasons of having two pointers rather than one is that in singly-linked list one does not have a reference to the precedent node, therefore we keep an additional pointer which points to the precedent node.

For this problem, we iterate through the cyclic list using two pointers, namely prev and curr. When we find a suitable place to insert the new value, we insert it between the prev and curr nodes.



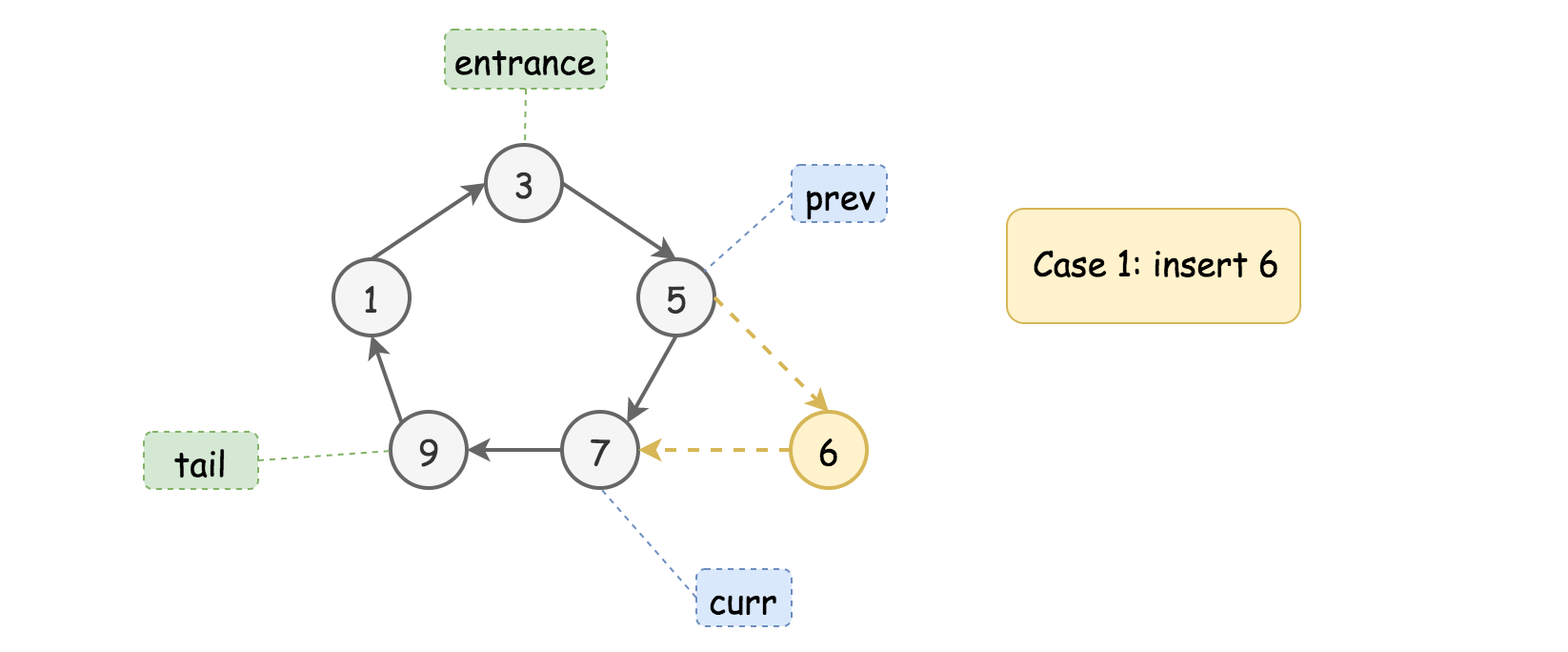
**Algorithm**

First of all, let us define the skeleton of two-pointers iteration algorithm as follows:

* As we mentioned in the intuition, we loop over the linked list with two pointers (i.e. prev and curr) step by step. The termination condition of the loop is that we get back to the starting point of the two pointers (i.e. prev == head)
* During the loop, at each step, we check if the current place bounded by the two pointers is the right place to insert the new value.
* If not, we move both pointers one step forwards.

Now, the tricky part of this problem is to sort out different cases that our algorithm should deal with within the loop, and then design a concise logic to handle them sound and properly. Here we break it down into three general cases.

**Case 1).** The value of new node sits between the minimal and maximal values of the current list. As a result, it should be inserted within the list.



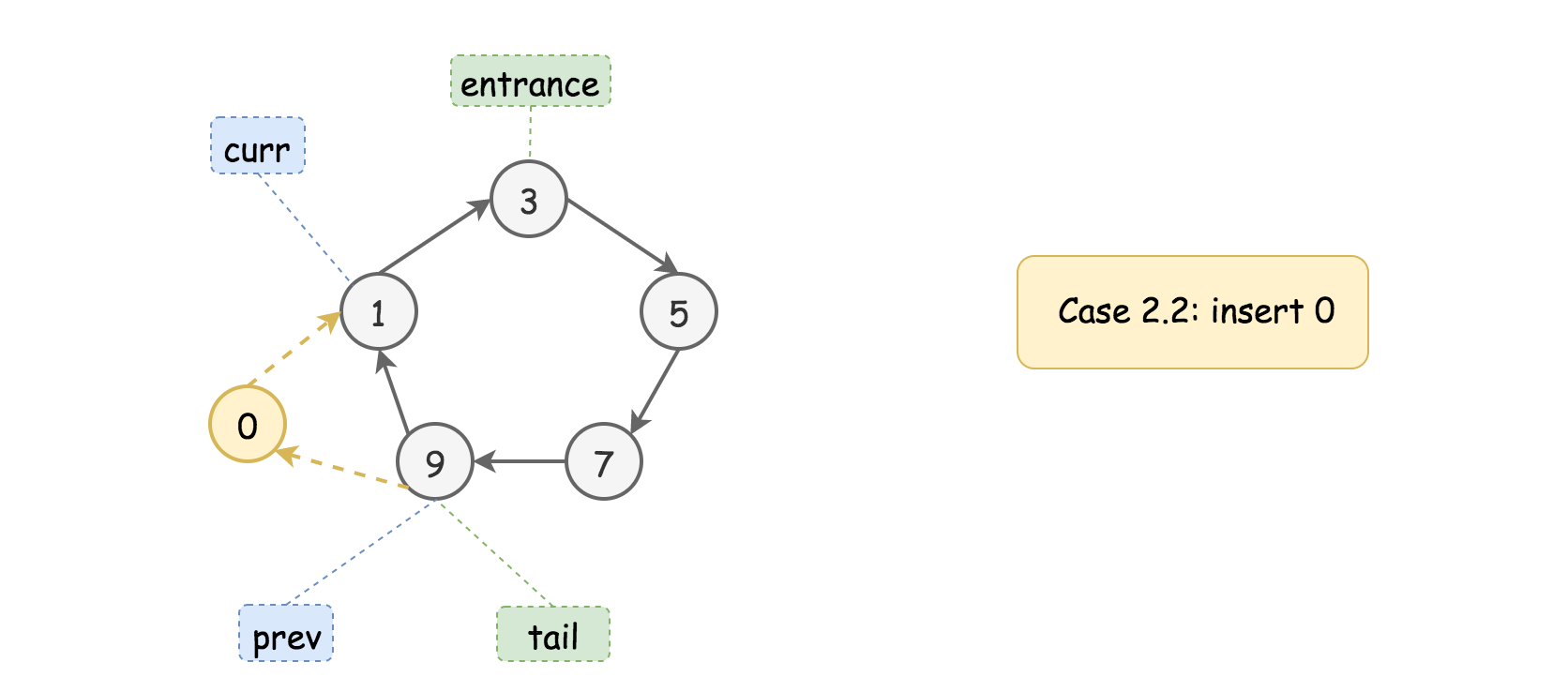
As we can see from the above example, the new value (6) sits between the minimal and maximal values of the list (i.e. 1 and 9). No matter where we start from (in this example we start from the node {3}), the new node would end up being inserted between the nodes {5} and {7}.

The condition is to find the place that meets the constraint of *{prev.val <= insertVal <= curr.val}*.

**Case 2).** The value of new node goes beyond the minimal and maximal values of the current list, either less than the minimal value or greater than the maximal value. In either case, the new node should be added right after the tail node (i.e. the node with the maximal value of the list).

Here are the examples with the same input list as in the previous example.





Firstly, we should locate the position of the **tail** node, by finding a descending order between the adjacent, i.e. the condition of {prev.val > curr.val}, since the nodes are sorted in ascending order, the tail node would have the greatest value of all nodes.

Furthermore, we check if the new value goes beyond the values of tail and head nodes, which are pointed by the prev and curr pointers respectively.

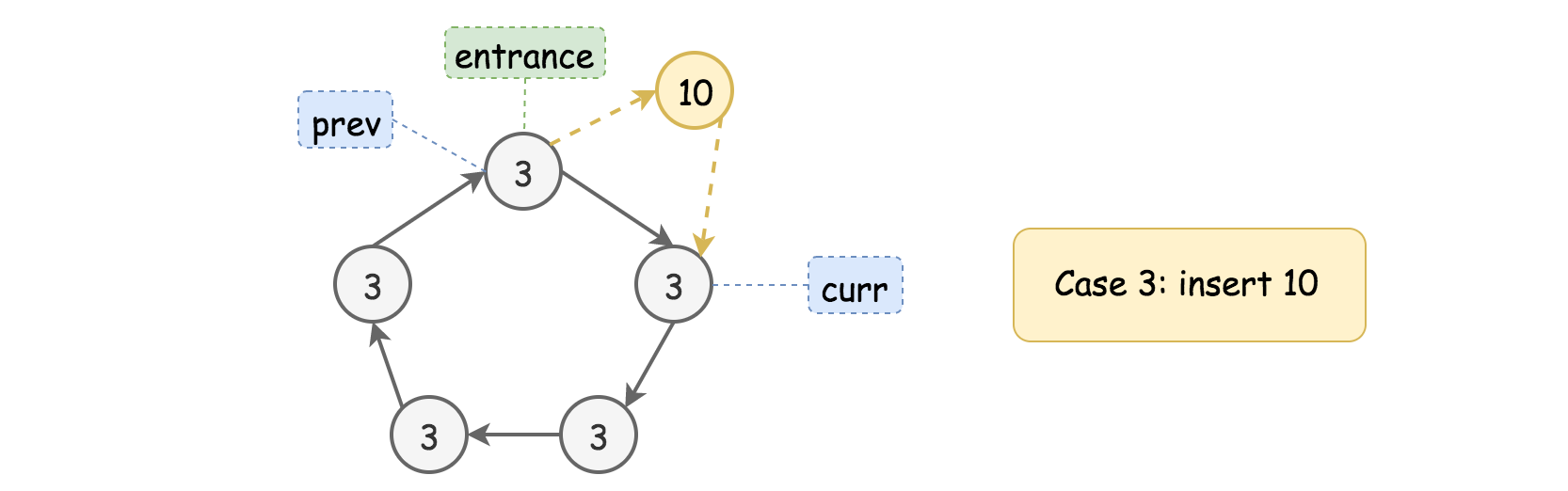
The Case 2.1 corresponds to the condition where the value to be inserted is greater than or equal to the one of tail node, i.e. {insertVal >= prev.val}.

The Case 2.2 corresponds to the condition where the value to be inserted is less than or equal to the head node, i.e. {insertVal <= curr.val}.

Once we locate the tail and head nodes, we basically **extend** the original list by inserting the value in between the tail and head nodes, i.e. in between the prev and curr pointers, the same operation as in the Case 1.

Case 3). Finally, there is one case that does not fall into any of the above two cases. This is the case where the list contains uniform values.

Though not explicitly stated in the problem description, our sorted list can contain some duplicate values. And in the extreme case, the entire list has only one single unique value.



In this case, we would end up looping through the list and getting back to the starting point.

The followup action is just to add the new node after any node in the list, regardless the value to be inserted. Since we are back to the starting point, we might as well add the new node right after the starting point (our entrance node).

Note that, we cannot skip the iteration though, since we have to iterate through the list to determine if our list contains a single unique value.

The above three cases cover the scenarios within and after our iteration loop. There is however one minor **corner** case we still need to deal with, where we have an **empty** list. This, we could easily handle before the loop.

|  |
| --- |
| class Solution {  public Node insert(Node head, int insertVal) {  if (head == null) {  Node newNode = new Node(insertVal, null);  newNode.next = newNode;  return newNode;  }  Node prev = head;  Node curr = head.next;  boolean toInsert = false;  do {  if (prev.val <= insertVal && insertVal <= curr.val) {  // Case 1).  toInsert = true;  } else if (prev.val > curr.val) {  // Case 2).  if (insertVal >= prev.val || insertVal <= curr.val)  toInsert = true;  }  if (toInsert) {  prev.next = new Node(insertVal, curr);  return head;  }  prev = curr;  curr = curr.next;  } while (prev != head);  // Case 3).  prev.next = new Node(insertVal, curr);  return head;  }  } |

**Complexity Analysis**

* Time Complexity: O(*N*) where N*N* is the size of list. In the worst case, we would iterate through the entire list.
* Space Complexity: O(1). It is a constant space solution.

**Copy List with Random Pointer**

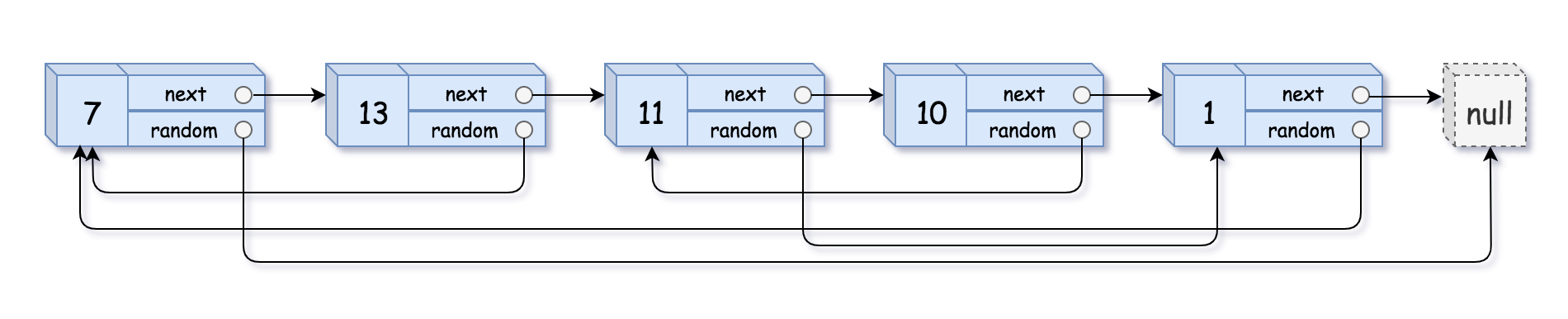
A linked list is given such that each node contains an additional random pointer which could point to any node in the list or null.

Return a [**deep copy**](https://en.wikipedia.org/wiki/Object_copying#Deep_copy) of the list.

The Linked List is represented in the input/output as a list of n nodes. Each node is represented as a pair of [val, random\_index] where:

* val: an integer representing Node.val
* random\_index: the index of the node (range from 0 to n-1) where random pointer points to, or null if it does not point to any node.

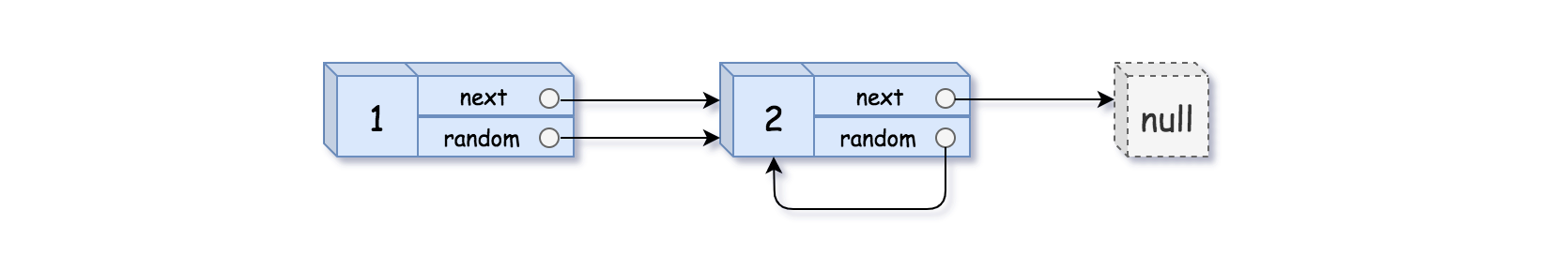
**Example 1:**



**Input:** head = [[7,null],[13,0],[11,4],[10,2],[1,0]]

**Output:** [[7,null],[13,0],[11,4],[10,2],[1,0]]

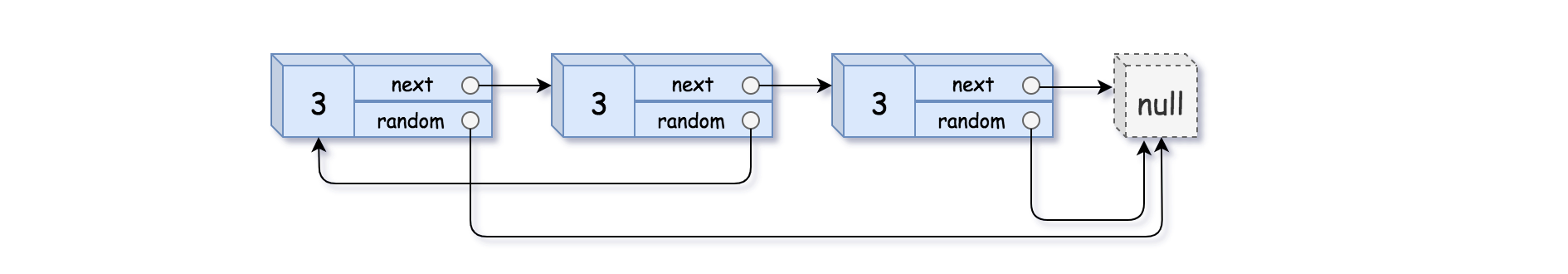
**Example 2:**



**Input:** head = [[1,1],[2,1]]

**Output:** [[1,1],[2,1]]

**Example 3:**

****

**Input:** head = [[3,null],[3,0],[3,null]]

**Output:** [[3,null],[3,0],[3,null]]

**Example 4:**

**Input:** head = []

**Output:** []

**Explanation:** Given linked list is empty (null pointer), so return null.

**Constraints:**

* -10000 <= Node.val <= 10000
* Node.random is null or pointing to a node in the linked list.
* The number of nodes will not exceed 1000.

Hint #1

Just iterate the linked list and create copies of the nodes on the go. Since a node can be referenced from multiple nodes due to the random pointers, make sure you are not making multiple copies of the same node.

Hint #2

You may want to use extra space to keep **old node ---> new node** mapping to prevent creating multiples copies of same node.

Hint #3

We can avoid using extra space for old node ---> new node mapping, by tweaking the original linked list. Simply interweave the nodes of the old and copied list. For e.g.

Old List: A --> B --> C --> D

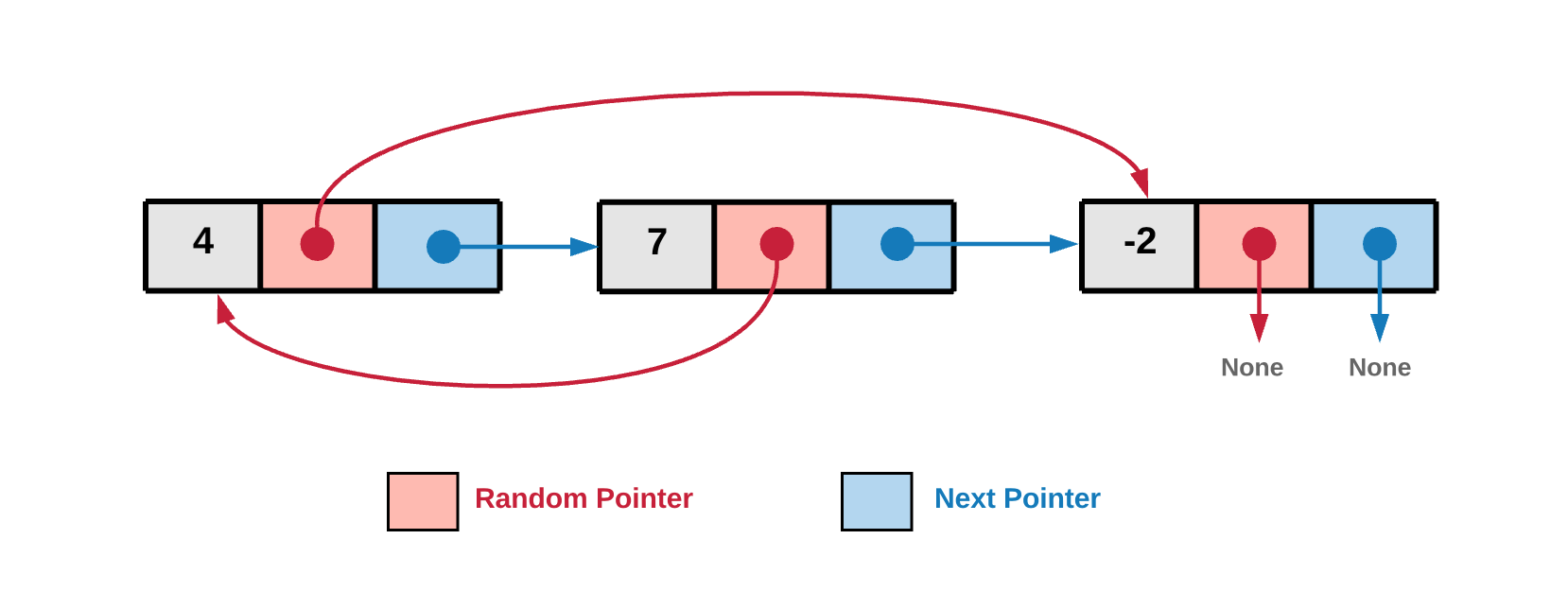
InterWeaved List: A --> A' --> B --> B' --> C --> C' --> D --> D'

Hint #4

The interweaving is done using **next** pointers and we can make use of interweaved structure to get the correct reference nodes for **random** pointers.

## Solution

Let’s first look at how the linked list looks like

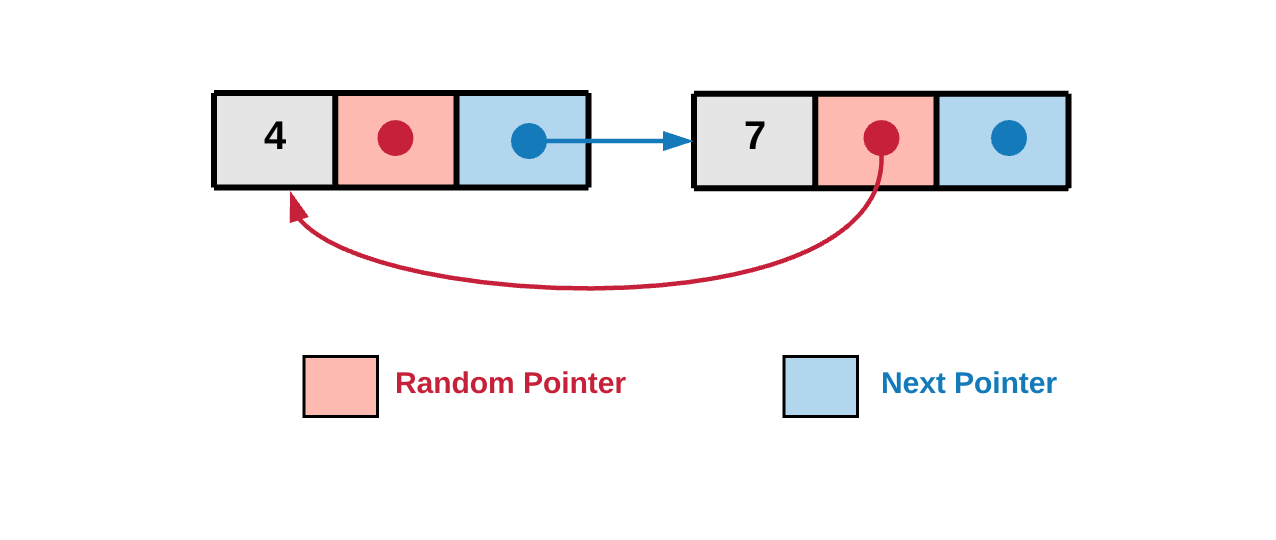


In the above diagram, for a given node the next pointer points to the next node in the linked list. The next pointer is something standard for a linked list and this is what ***links*** the nodes together. What is interesting about the diagram and this problem is the random pointer which, as the name suggests can point to any node in the linked list or can be a null.

#### **Approach 1: Recursive**

**Intuition**

The basic idea behind the recursive solution is to consider the linked list like a graph. Every node of the Linked List has 2 pointers (edges in a graph). Since, random pointers add the randomness to the structure we might visit the same node again leading to cycles.



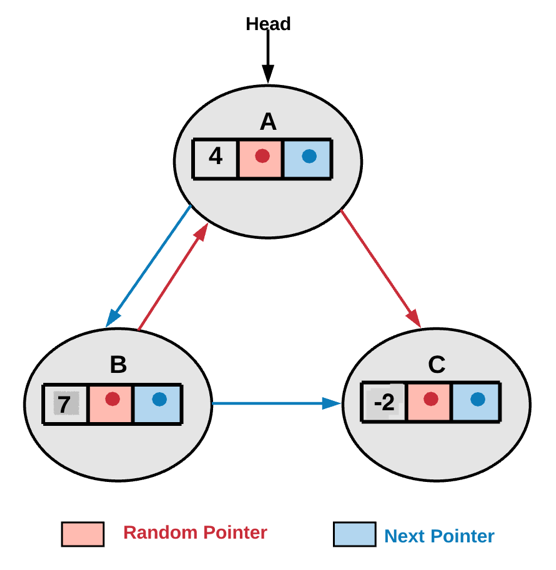
In the diagram above we can see the random pointer points back to the previously seen node hence leading to a cycle. We need to take care of these cycles in the implementation.

All we do in this approach is to just traverse the graph and clone it. Cloning essentially means creating a new node for every unseen node you encounter. The traversal part will happen recursively in a depth first manner. Note that we have to keep track of nodes already processed because, as pointed out earlier, we can have cycles because of the random pointers.

**Algorithm**

1. Start traversing the graph from head node.

Let’s see the linked structure as a graph. Below is the graph representation of the above linked list example.



In the above example head is where we begin our graph traversal.

1. If we already have a cloned copy of the current node in the visited dictionary, we use the cloned node reference.
2. If we don't have a cloned copy in the visited dictionary, we create a new node and add it to the visited dictionary. visited\_dictionary[current\_node] = cloned\_node\_for\_current\_node.
3. We then make two recursive calls, one using the random pointer and the other using next pointer. The diagram from step 1, shows random and next pointers in red and blue color respectively. Essentially we are making recursive calls for the children of the current node. In this implementation, the children are the nodes pointed by the random and the next pointers.

cloned\_node\_for\_current\_node.next = copyRandomList(current\_node.next);

cloned\_node\_for\_current\_node.random = copyRandomList(current\_node.random);

|  |
| --- |
| /\*  // Definition for a Node.  class Node {  public int val;  public Node next;  public Node random;  public Node() {}  public Node(int \_val,Node \_next,Node \_random) {  val = \_val;  next = \_next;  random = \_random;  }  };  \*/  public class Solution {  // HashMap which holds old nodes as keys and new nodes as its values.  HashMap<Node, Node> visitedHash = new HashMap<Node, Node>();  public Node copyRandomList(Node head) {  if (head == null) {  return null;  }  // If we have already processed the current node, then we simply return the cloned version of  // it.  if (this.visitedHash.containsKey(head)) {  return this.visitedHash.get(head);  }  // Create a new node with the value same as old node. (i.e. copy the node)  Node node = new Node(head.val, null, null);  // Save this value in the hash map. This is needed since there might be  // loops during traversal due to randomness of random pointers and this would help us avoid  // them.  this.visitedHash.put(head, node);  // Recursively copy the remaining linked list starting once from the next pointer and then from  // the random pointer.  // Thus we have two independent recursive calls.  // Finally we update the next and random pointers for the new node created.  node.next = this.copyRandomList(head.next);  node.random = this.copyRandomList(head.random);  return node;  }  } |

**Complexity Analysis**

* Time Complexity: *O*(*N*) where N is the number of nodes in the linked list.
* Space Complexity: *O*(*N*). If we look closely, we have the recursion stack and we also have the space complexity to keep track of nodes already cloned i.e. using the visited dictionary. But asymptotically, the complexity is *O*(*N*).

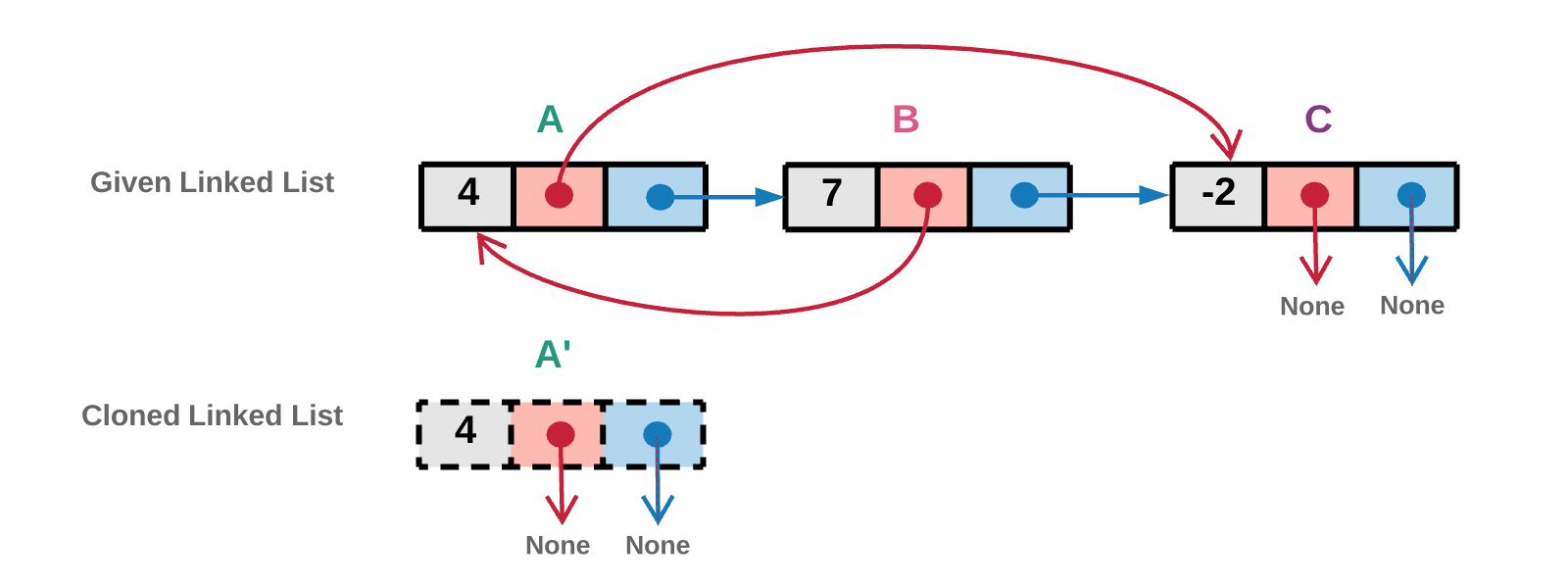
#### **Approach 2: Iterative with O(N) Space**

**Intuition**

The iterative solution to this problem does not model it as a graph, instead simply treats it as a LinkedList. When we are iterating over the list, we can create new nodes via the random pointer or the next pointer whichever points to a node that doesn't exist in our old --> new dictionary.

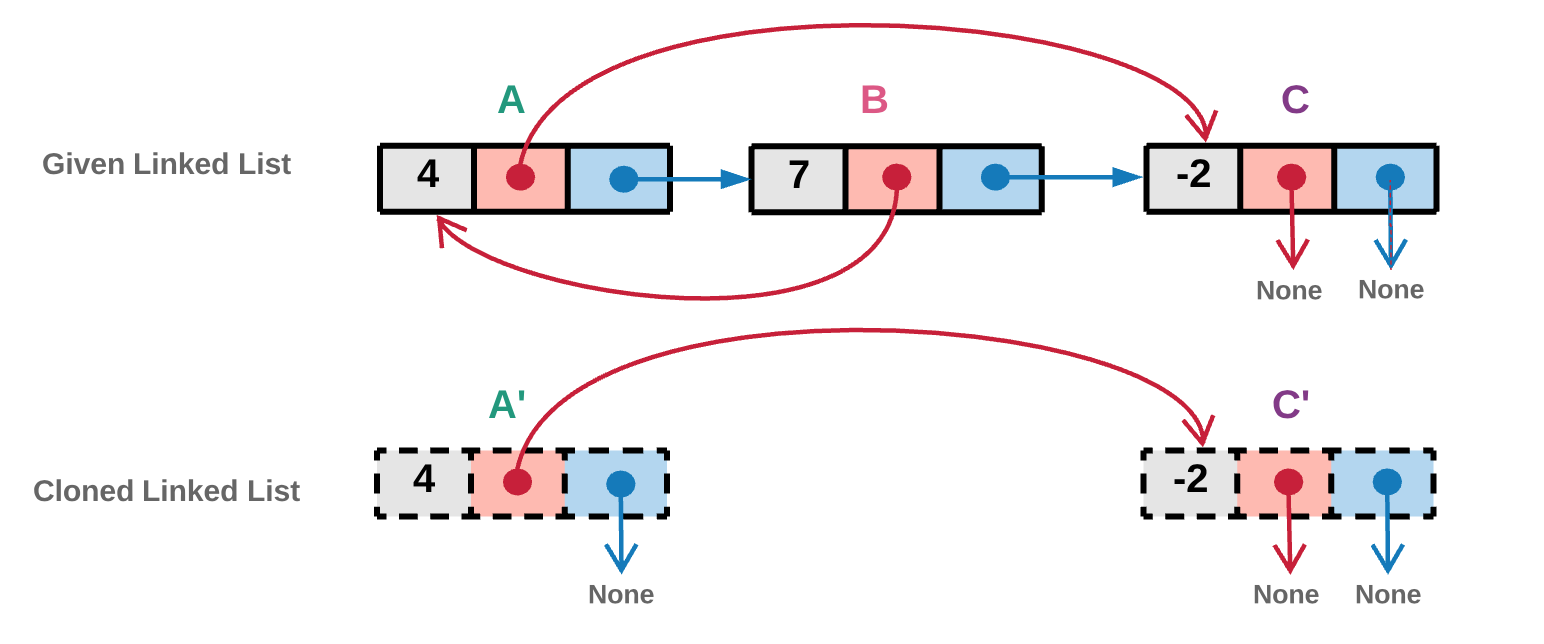
**Algorithm**

1. Traverse the linked list starting at head of the linked list.



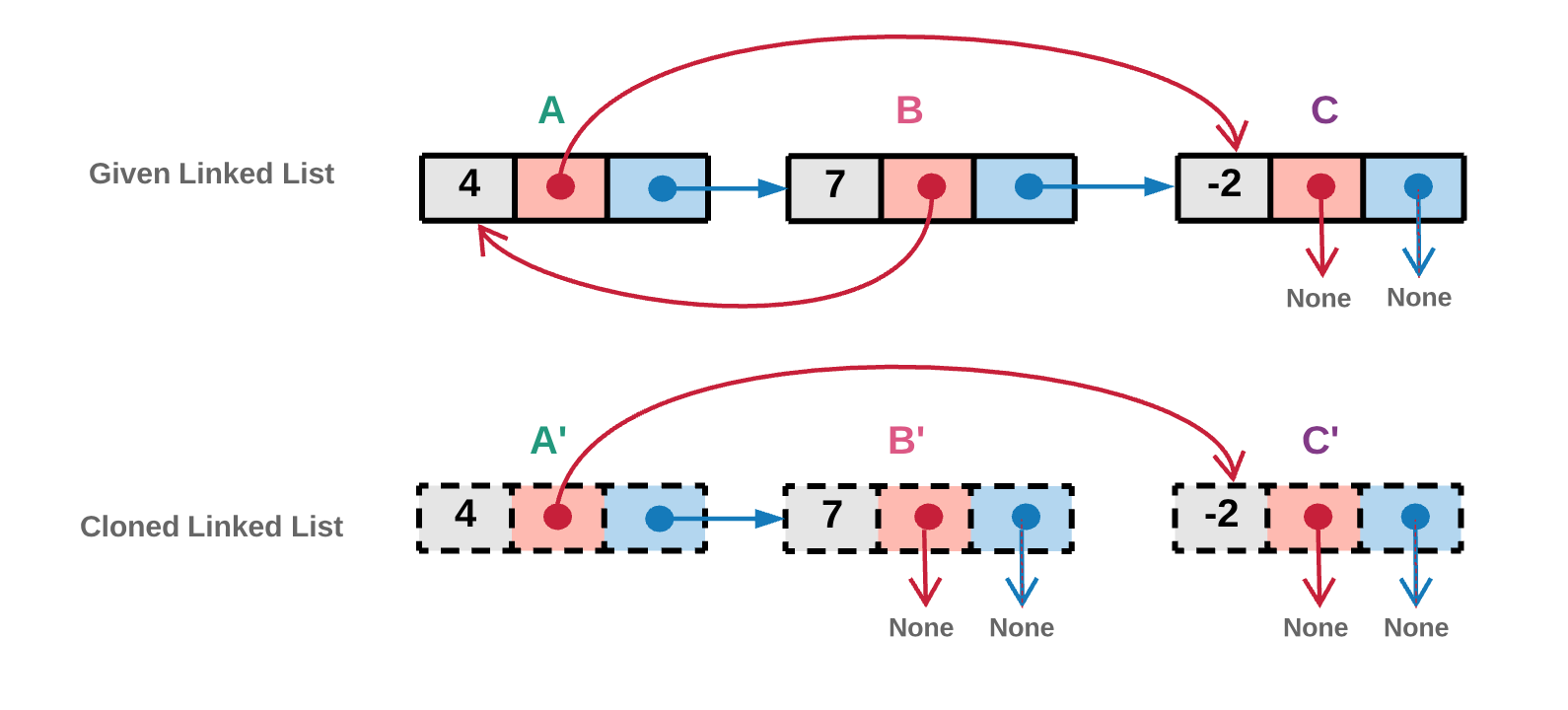
In the above diagram we create a new cloned head node. The cloned node is shown using dashed lines. In the implementation we would even store the reference of this newly created node in a visited dictionary.

1. Random Pointer
   * If the random pointer of the current node *i* points to the a node *j* and a clone of *j* already exists in the visited dictionary, we will simply use the cloned node reference from the visited dictionary.
   * If the random pointer of the current node *i* points to the a node *j* which has not been created yet, we create a new node corresponding to *j* and add it to the visited dictionary.



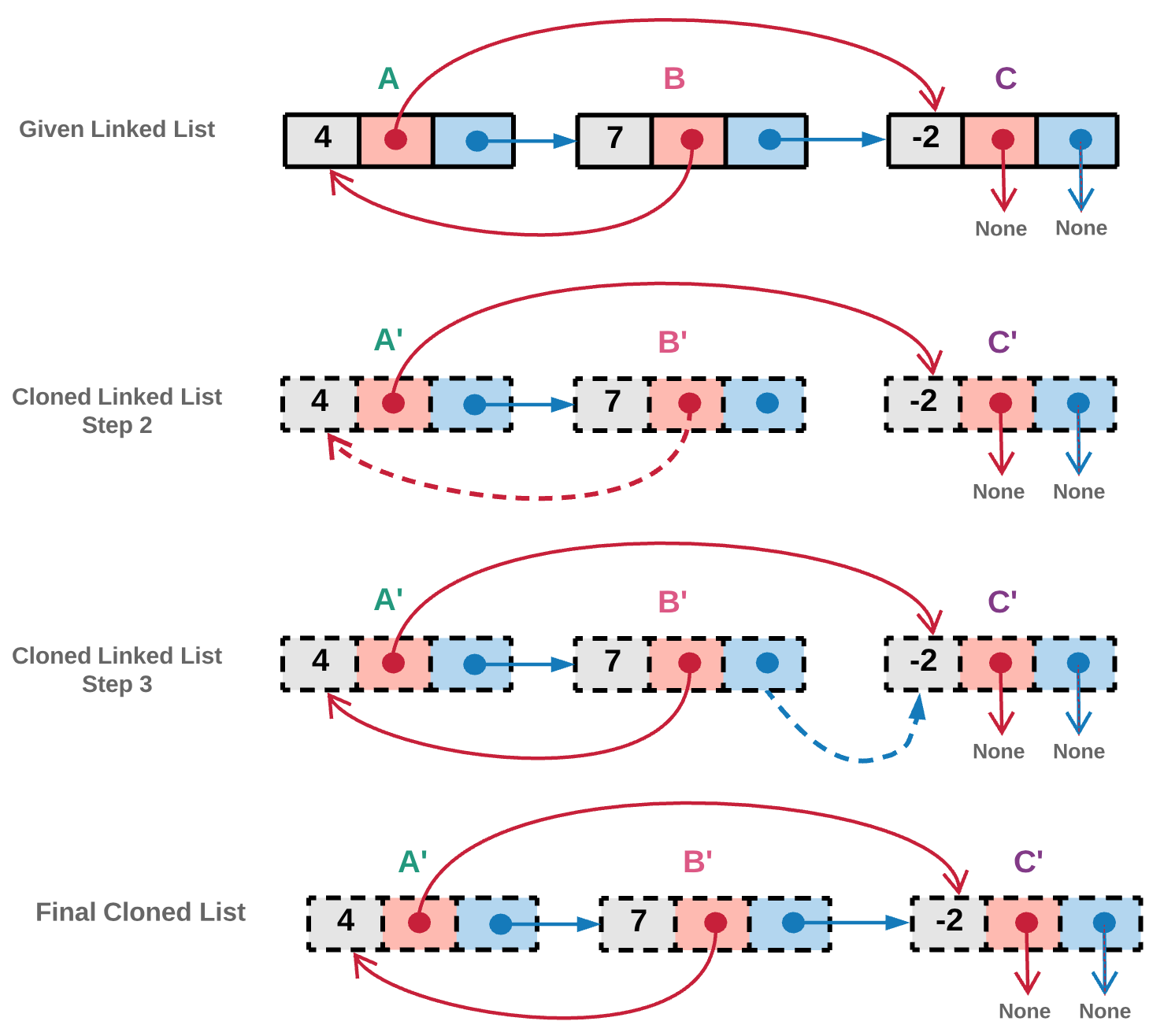
In the above diagram the random pointer of node *A* points to a node *C*. Node *C* which was not visited yet as we can see from the previous diagram. Hence we create a new cloned *C*′ node corresponding to node *C* and add it to visited dictionary.

1. Next Pointer
   * If the next pointer of the current node *i* points to the a node *j* and a clone of *j* already exists in the visited dictionary, we will simply use the cloned node reference from the visited dictionary.
   * If the next pointer of the current node *i* points to the a node *j* which has not been created yet, we create a new node corresponding to *j* and add it to the visited dictionary.



In the above diagram the next pointer of node *A* points to a node B*B*. Node *B* which was not visited yet as we can see from the previous diagram. Hence we create a new cloned *B*′ node corresponding to node *B* and add it to visited dictionary.

1. We repeat steps 2 and 3 until we reach the end of the linked list.



In the above diagram, the random pointer of node *B* points to an already visited node *A*. Hence in step 2, we don't create a new copy for the clone. Instead we point random pointer of cloned node *B*′ to already existing cloned node *A*′.

Also, the next pointer of node *B* points to an already visited node *C*. Hence in step 3, we don't create a new copy for the clone. Instead we point next pointer of cloned node *B*′ to already existing cloned node *C*′.

|  |
| --- |
| /\*  // Definition for a Node.  class Node {  public int val;  public Node next;  public Node random;  public Node() {}  public Node(int \_val,Node \_next,Node \_random) {  val = \_val;  next = \_next;  random = \_random;  }  };  \*/  public class Solution {  // Visited dictionary to hold old node reference as "key" and new node reference as the "value"  HashMap<Node, Node> visited = new HashMap<Node, Node>();  public Node getClonedNode(Node node) {  // If the node exists then  if (node != null) {  // Check if the node is in the visited dictionary  if (this.visited.containsKey(node)) {  // If its in the visited dictionary then return the new node reference from the dictionary  return this.visited.get(node);  } else {  // Otherwise create a new node, add to the dictionary and return it  this.visited.put(node, new Node(node.val, null, null));  return this.visited.get(node);  }  }  return null;  }  public Node copyRandomList(Node head) {  if (head == null) {  return null;  }  Node oldNode = head;  // Creating the new head node.  Node newNode = new Node(oldNode.val);  this.visited.put(oldNode, newNode);  // Iterate on the linked list until all nodes are cloned.  while (oldNode != null) {  // Get the clones of the nodes referenced by random and next pointers.  newNode.random = this.getClonedNode(oldNode.random);  newNode.next = this.getClonedNode(oldNode.next);  // Move one step ahead in the linked list.  oldNode = oldNode.next;  newNode = newNode.next;  }  return this.visited.get(head);  }  } |

**Complexity Analysis**

* Time Complexity : *O*(*N*) because we make one pass over the original linked list.
* Space Complexity : *O*(*N*) as we have a dictionary containing mapping from old list nodes to new list nodes. Since there are *N* nodes, we have *O*(*N*) space complexity.

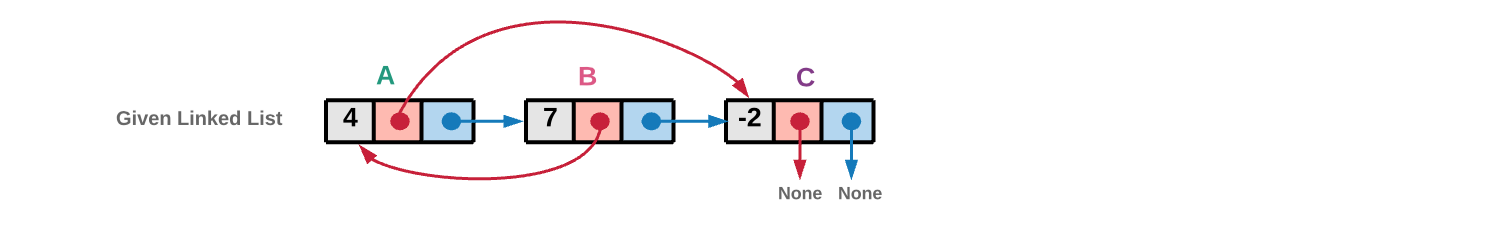
#### **Approach 3: Iterative with O(1) Space**

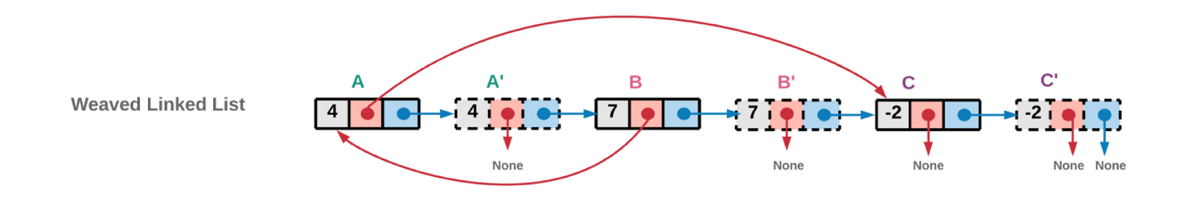
**Intuition**

Instead of a separate dictionary to keep the old node --> new node mapping, we can tweak the original linked list and keep every cloned node next to its original node. This interleaving of old and new nodes allows us to solve this problem without any extra space. Lets look at how the algorithm works.

**Algorithm**

1. Traverse the original list and clone the nodes as you go and place the cloned copy next to its original node. This new linked list is essentially a interweaving of original and cloned nodes.



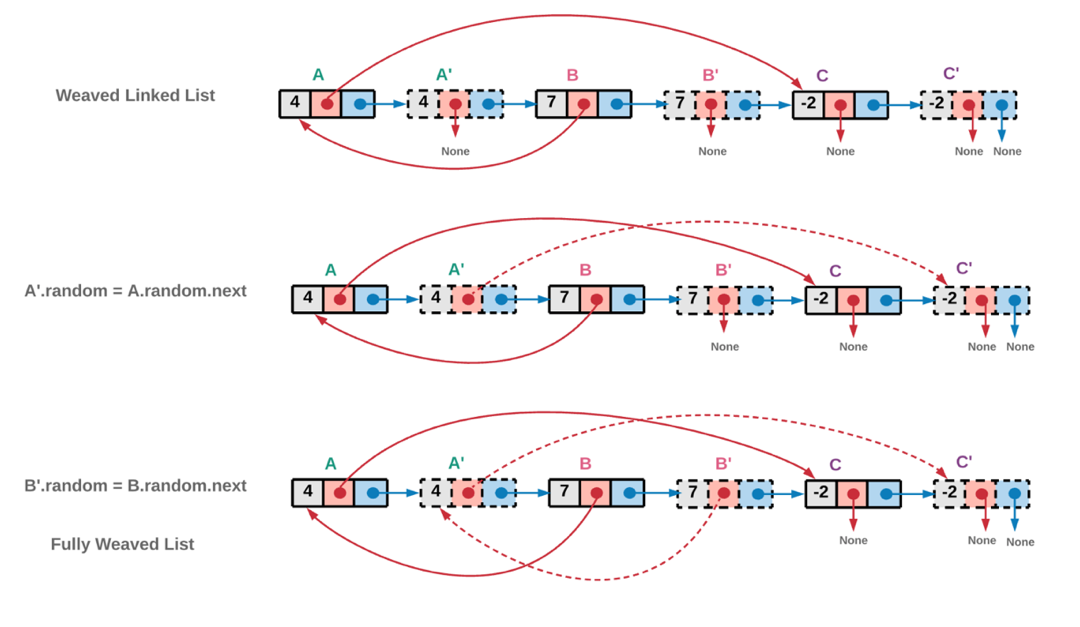


As you can see we just use the value of original node to create the cloned copy. The next pointer is used to create the weaving. Note that this operation ends up modifying the original linked list.

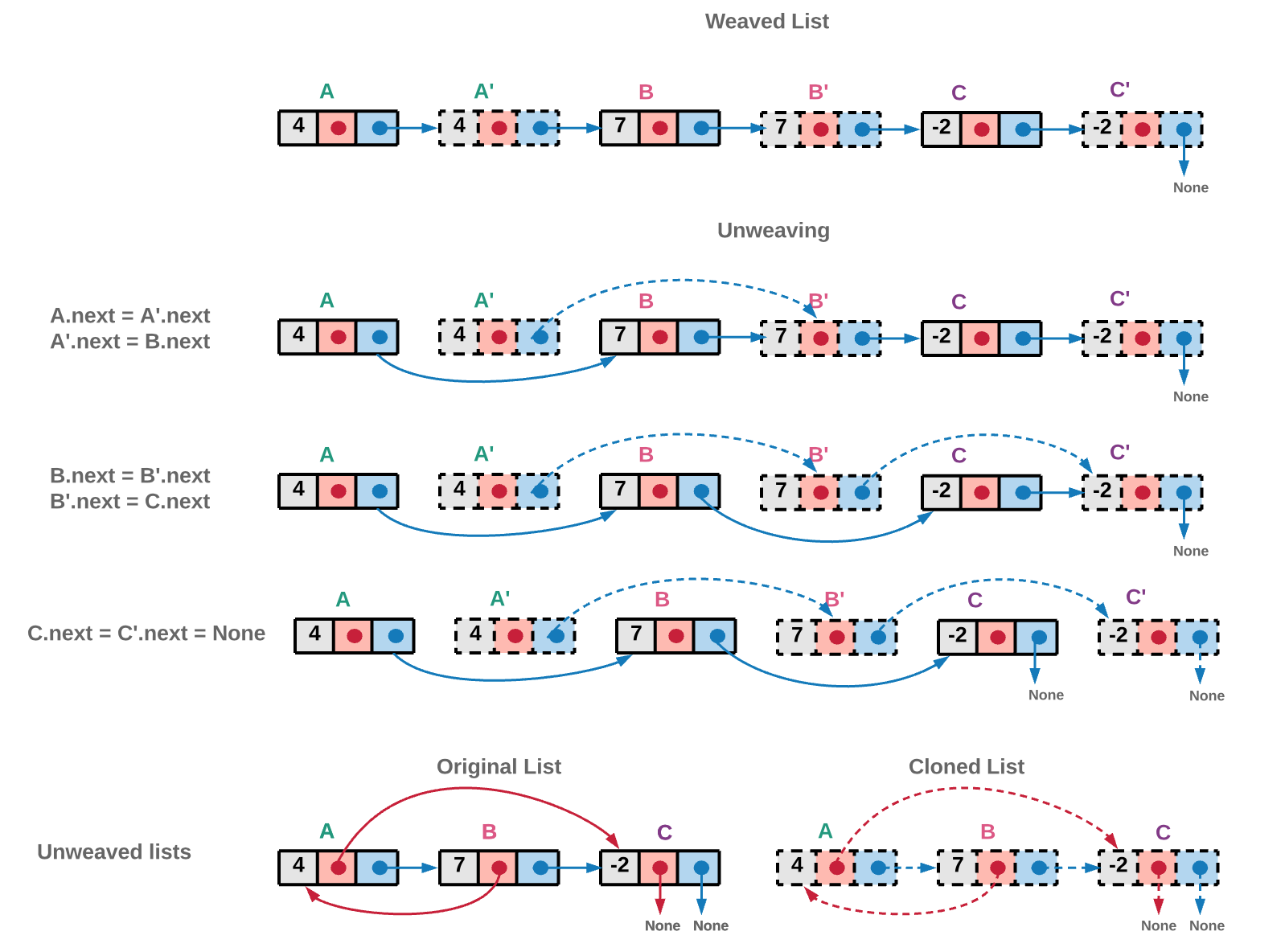
cloned\_node.next = original\_node.next

original\_node.next = cloned\_node

1. Iterate the list having both the new and old nodes intertwined with each other and use the original nodes' random pointers to assign references to random pointers for cloned nodes. For eg. If B has a random pointer to A, this means B' has a random pointer to A'.



1. Now that the random pointers are assigned to the correct node, the next pointers need to be correctly assigned to unweave the current linked list and get back the original list and the cloned list.



|  |
| --- |
| /\*  // Definition for a Node.  class Node {  public int val;  public Node next;  public Node random;  public Node() {}  public Node(int \_val,Node \_next,Node \_random) {  val = \_val;  next = \_next;  random = \_random;  }  };  \*/  public class Solution {  public Node copyRandomList(Node head) {  if (head == null) {  return null;  }  // Creating a new weaved list of original and copied nodes.  Node ptr = head;  while (ptr != null) {  // Cloned node  Node newNode = new Node(ptr.val);  // Inserting the cloned node just next to the original node.  // If A->B->C is the original linked list,  // Linked list after weaving cloned nodes would be A->A'->B->B'->C->C'  newNode.next = ptr.next;  ptr.next = newNode;  ptr = newNode.next;  }  ptr = head;  // Now link the random pointers of the new nodes created.  // Iterate the newly created list and use the original nodes' random pointers,  // to assign references to random pointers for cloned nodes.  while (ptr != null) {  ptr.next.random = (ptr.random != null) ? ptr.random.next : null;  ptr = ptr.next.next;  }  // Unweave the linked list to get back the original linked list and the cloned list.  // i.e. A->A'->B->B'->C->C' would be broken to A->B->C and A'->B'->C'  Node ptr\_old\_list = head; // A->B->C  Node ptr\_new\_list = head.next; // A'->B'->C'  Node head\_old = head.next;  while (ptr\_old\_list != null) {  ptr\_old\_list.next = ptr\_old\_list.next.next;  ptr\_new\_list.next = (ptr\_new\_list.next != null) ? ptr\_new\_list.next.next : null;  ptr\_old\_list = ptr\_old\_list.next;  ptr\_new\_list = ptr\_new\_list.next;  }  return head\_old;  }  } |

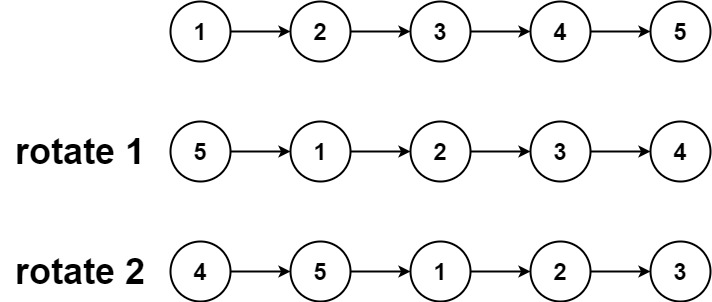
**Complexity Analysis**

* Time Complexity : *O*(*N*)
* Space Complexity : *O*(1)

**Rotate List**

Given the head of a linked list, rotate the list to the right by k places.

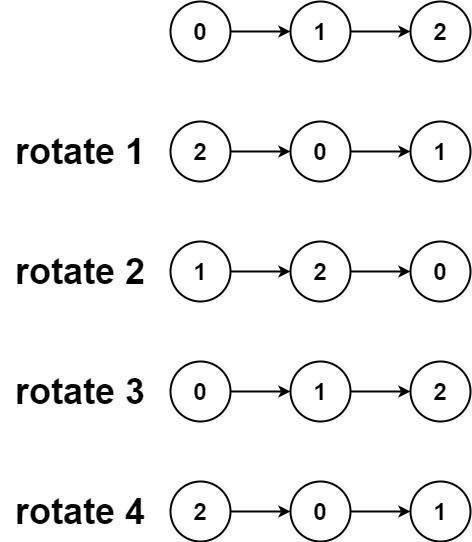
**Example 1:**



**Input:** head = [1,2,3,4,5], k = 2

**Output:** [4,5,1,2,3]

**Example 2:**



**Input:** head = [0,1,2], k = 4

**Output:** [2,0,1]

**Constraints:**

* The number of nodes in the list is in the range [0, 500].
* -100 <= Node.val <= 100
* 0 <= k <= 2 \* 109

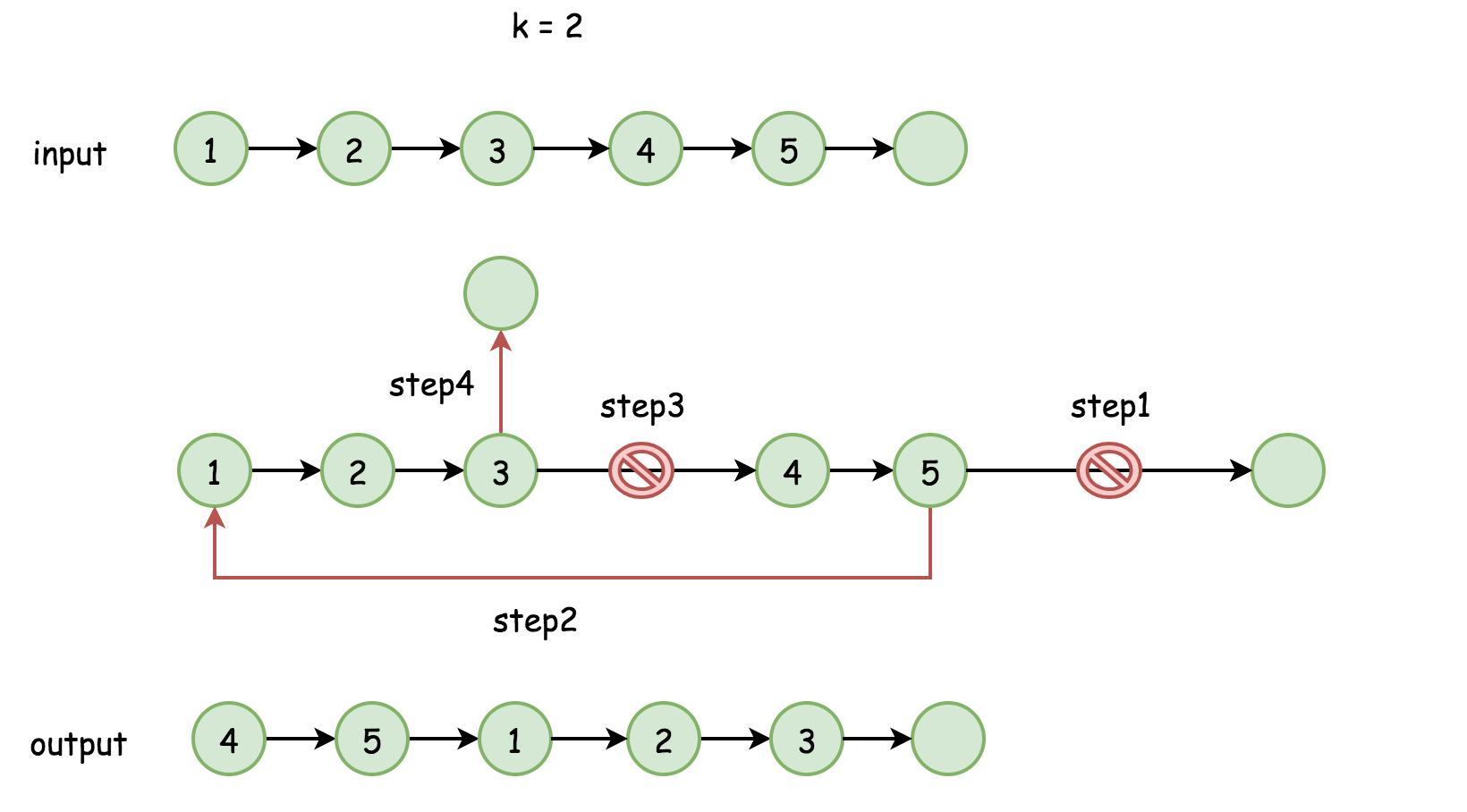
## Solution

#### **Approach 1:**

**Intuition**

The nodes in the list are already linked, and hence the rotation basically means

* To close the linked list into the ring.
* To break the ring after the new tail and just in front of the new head.



Where is the new head?

In the position n - k, where n is a number of nodes in the list. The new tail is just before, in the position n - k - 1.

We were assuming that k < n. What about the case of k >= n?

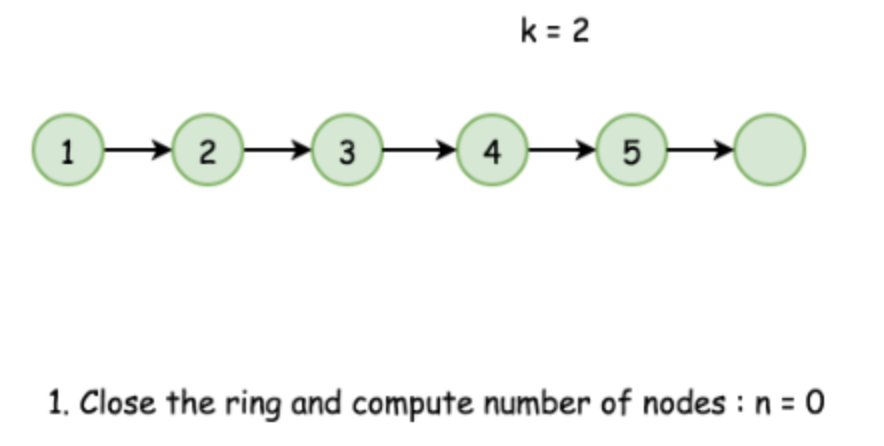
k could be rewritten as a sum k = (k // n) \* n + k % n, where the first term doesn't result in any rotation. Hence one could simply replace k by k % n to always have number of rotation places smaller than n.

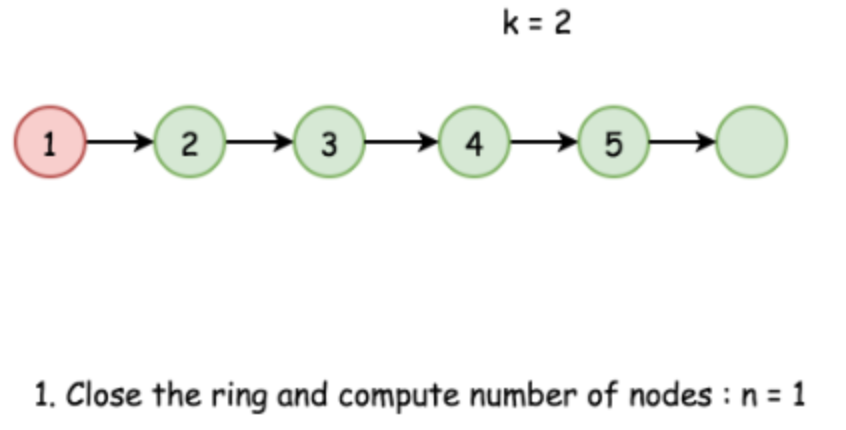
**Algorithm**

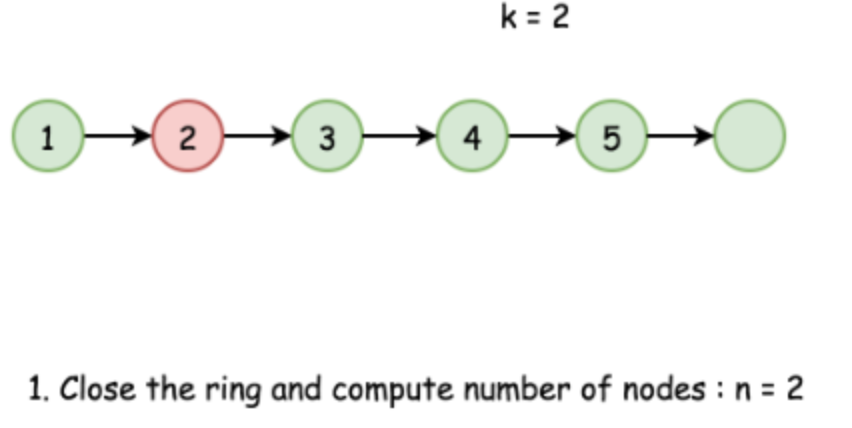
The algorithm is quite straightforward :

* Find the old tail and connect it with the head old\_tail.next = head to close the ring. Compute the length of the list n at the same time.
* Find the new tail, which is (n - k % n - 1)th node from the head and the new head, which is (n - k % n)th node.
* Break the ring new\_tail.next = None and return new\_head.

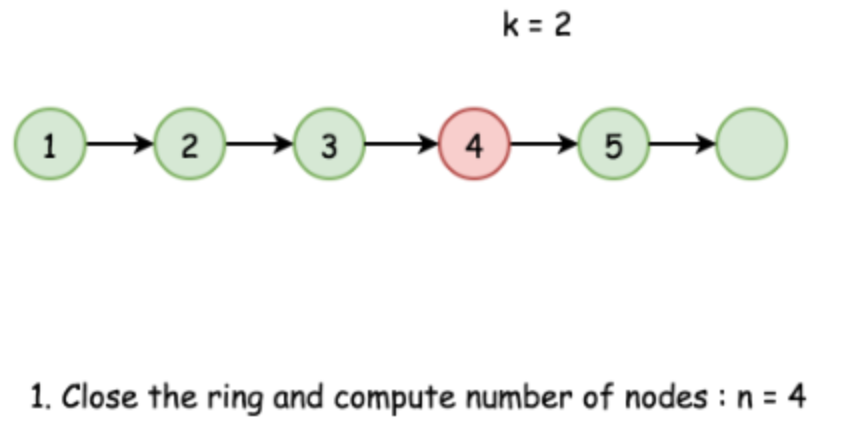
**Implementation**

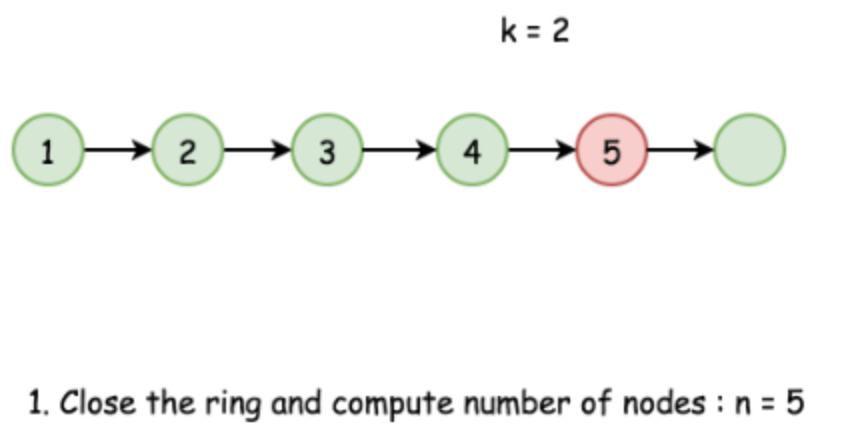


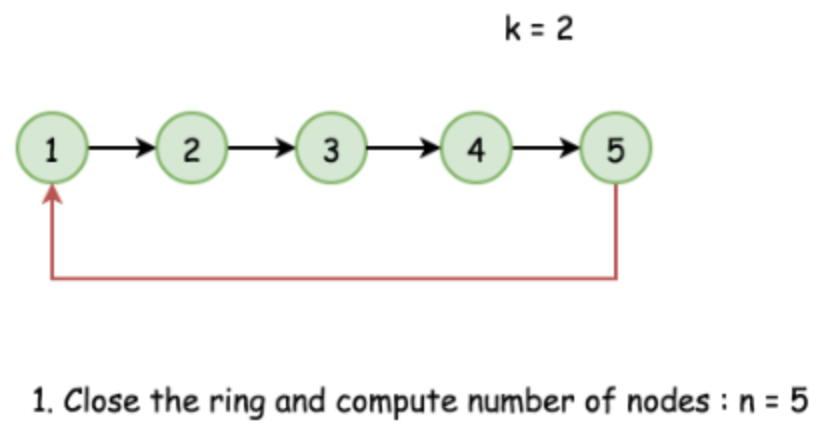


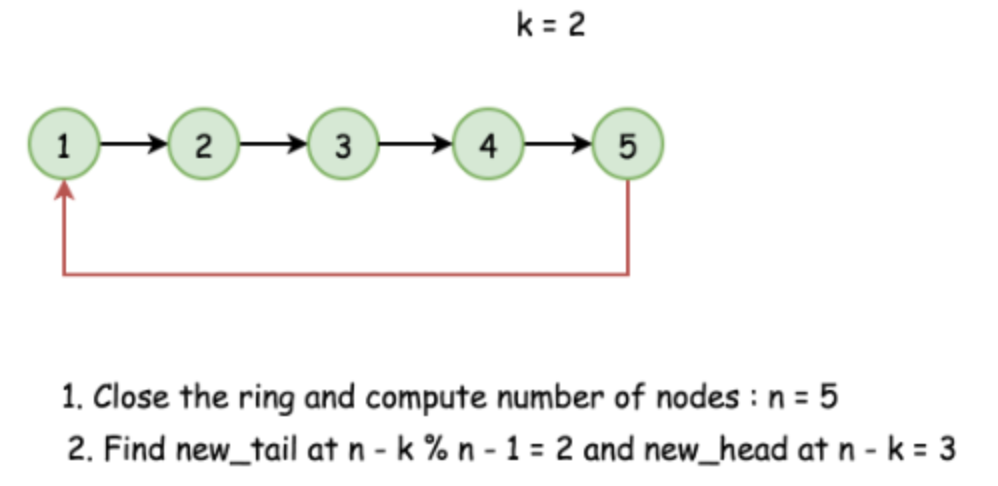


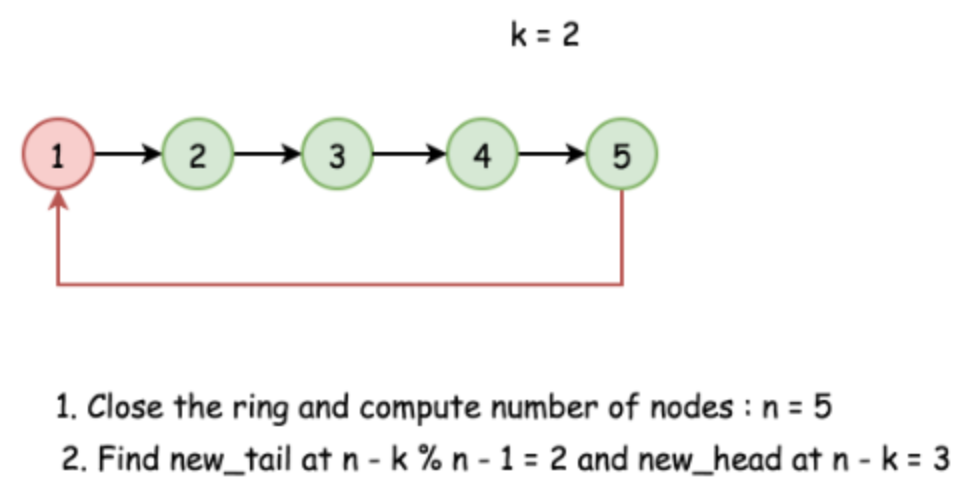


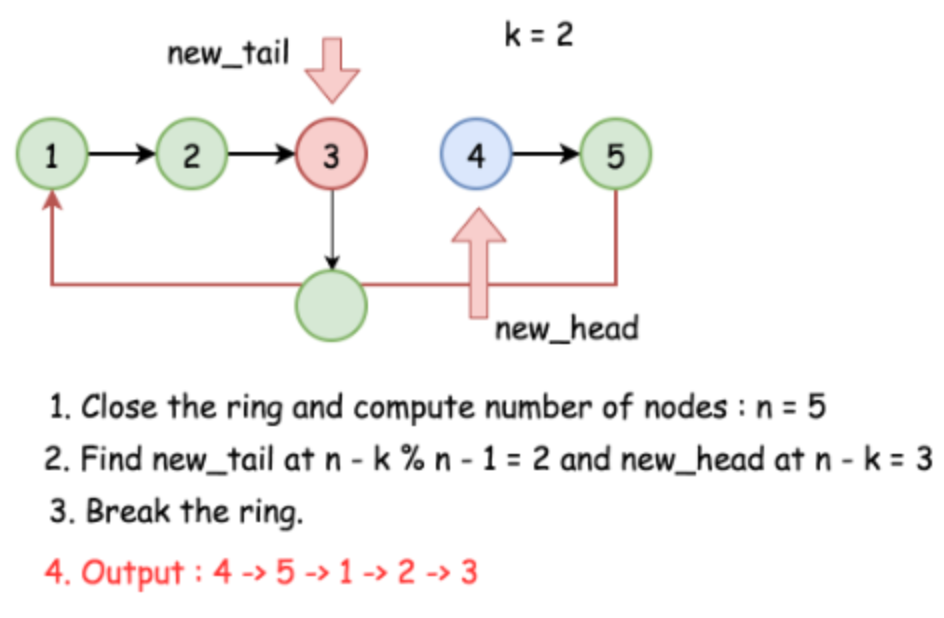
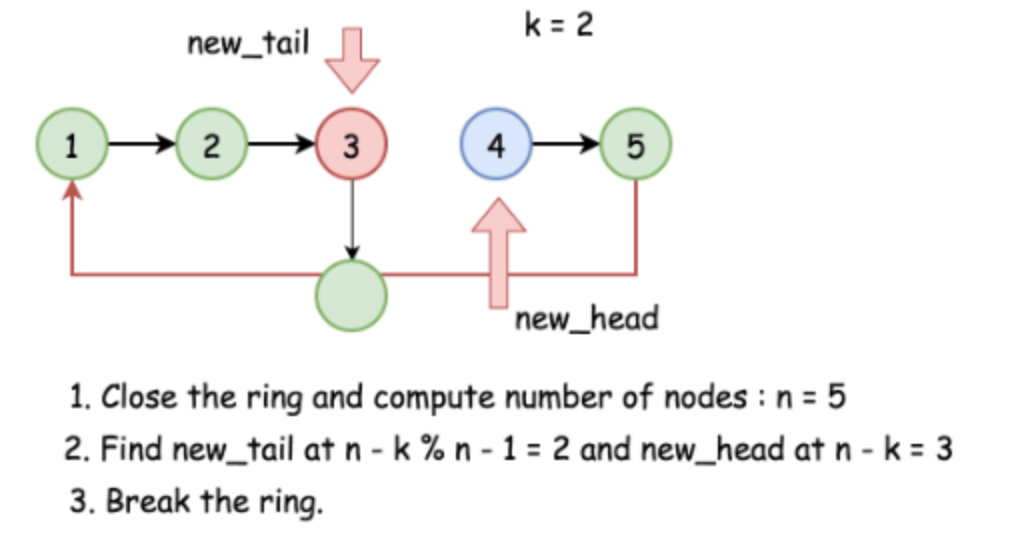
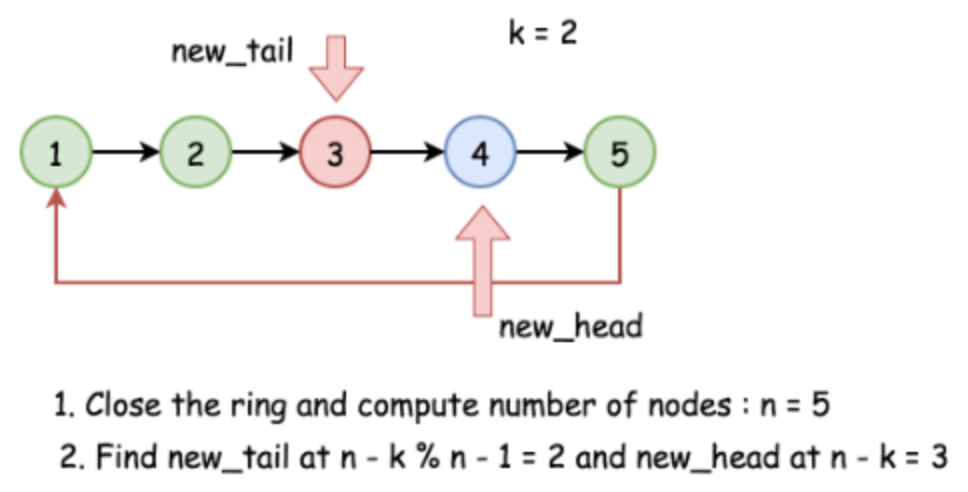
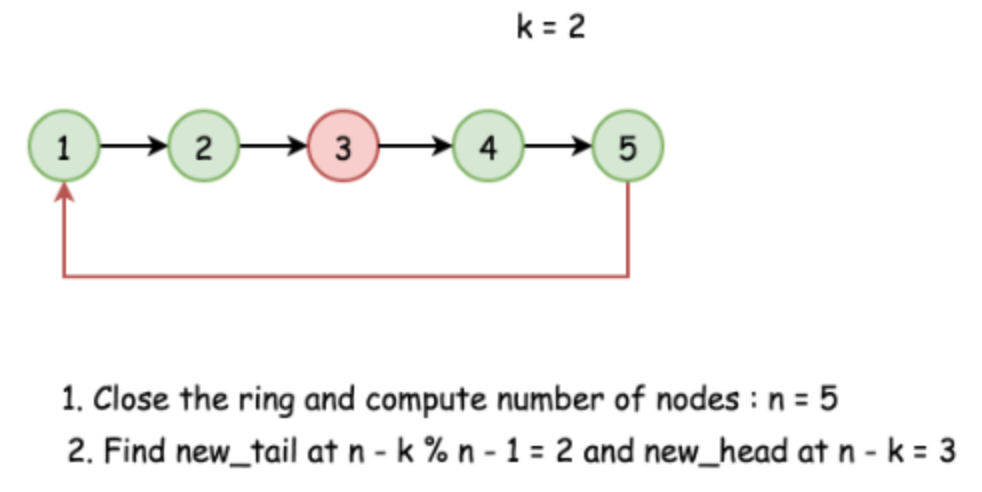
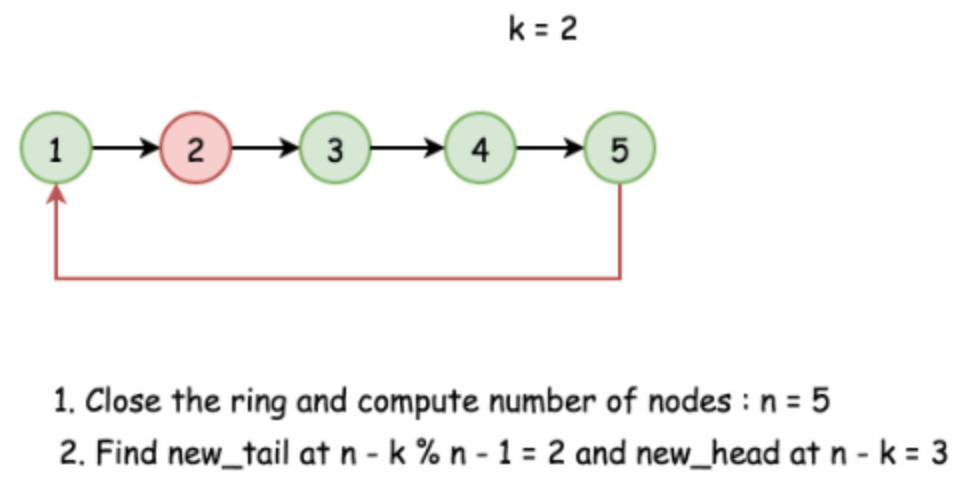












|  |
| --- |
| class Solution {  public ListNode rotateRight(ListNode head, int k) {  // base cases  if (head == null) return null;  if (head.next == null) return head;  // close the linked list into the ring  ListNode old\_tail = head;  int n;  for(n = 1; old\_tail.next != null; n++)  old\_tail = old\_tail.next;  old\_tail.next = head;  // find new tail : (n - k % n - 1)th node  // and new head : (n - k % n)th node  ListNode new\_tail = head;  for (int i = 0; i < n - k % n - 1; i++)  new\_tail = new\_tail.next;  ListNode new\_head = new\_tail.next;  // break the ring  new\_tail.next = null;  return new\_head;  }  } |

**Complexity Analysis**

* Time complexity : O(*N*) where *N* is a number of elements in the list.
* Space complexity : O(1) since it's a constant space solution.