与Reverse LinkedList有关的

# 203. 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

## 算法1：

思路：对于遇到相等的情况，就是利用当前节点的前一节点的next指向当前节点的后一节点。但是由于head不存在前一节点，因此先对head节点单独处理，处理之后head就可以充当prevNode了。

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

public ListNode removeElements(ListNode head, int val) {

//由于head不存在prevNode，所以先处理第一个节点，保证下一个节点的prevNode存在

//鉴于此，可以认为补充一个节点fakeNode充当head的prevNode，这样就可以利用while循环判断head了（见升级版）

while(head != null && head.val == val){//若开头相等，则循环删除开头元素

head = head.next;

}

if(head == null){

return head;

}

ListNode head1 = head.next;//保存需要返回的头head

ListNode prevNode = head;//上一个节点

while(head1 != null){

ListNode nextNodeTemp = head1.next;

**if(head1.val == val){**

**//若相等，改变prevNode的next属性，而prevNode不变**

**prevNode.next = nextNodeTemp != null ? nextNodeTemp : null;**

**}else{**

**prevNode = head1;**

**}**

**head1 = nextNodeTemp;//遍历一遍**

}

return head;

}

}

## 算法2：补充假节点fakeNode

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

public ListNode removeElements(ListNode head, int val) {

//由于head不存在prevNode，所以先处理第一个节点，保证下一个节点的prevNode存在

//鉴于此，可以认为补充一个节点fakeNode充当head的prevNode，这样就可以利用while循环判断head了

ListNode fakeNode = new ListNode(-1);//假节点

fakeNode.next = head;

ListNode prevNode = fakeNode;//上一个节点

while(head != null){

ListNode nextNodeTemp = head.next;

if(head.val == val){

//若相等，改变prevNode的next属性，而prevNode不变

prevNode.next = nextNodeTemp != null ? nextNodeTemp : null;

}else{

prevNode = head;

}

head = nextNodeTemp;//遍历一遍

}

return fakeNode.next;//利用fakeNode.next作为返回的头head

}

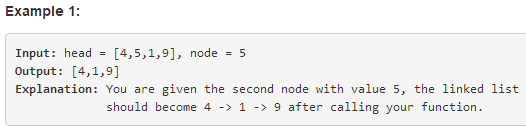
}

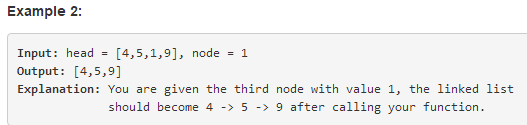
# 237. Delete Node in a Linked List

Write a function to delete a node (except the tail) in a singly linked list, given only access to that node.

Given linked list -- head = [4,5,1,9], which looks like following:

4 -> 5 -> 1 -> 9





Note:

* The linked list will have at least two elements.
* All of the nodes' values will be unique.
* The given node will not be the tail and it will always be a valid node of the linked list.
* Do not return anything from your function.

## 算法：

思路：//由于无法获取上一个节点，且题意说明了任意节点的值都是唯一的，也就是说只删除当前节点；但是直接删除当前节点需要修改上一节点的next，这里无法获取上一个节点，因此，先拷贝后一个节点到当前节点，然后再删除后一个节点即可实现。

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

public void deleteNode(ListNode node) {

//由于无法获取上一个节点，且题意说明了任意节点的值都是唯一的，也就是说只删除当前节点

//但是直接删除当前节点需要修改上一节点的next，这里无法获取上一个节点，因此，

//先拷贝后一个节点到当前节点，然后再删除后一个节点即可实现。

if(node == null) return;

if(node.next != null){

node.val = node.next.val;

node.next = node.next.next;

}

}

}

# 206. Reverse Linked List

(难度：easy)

**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?

## 算法1(比较简单)：Excellent

思路：需要两个外加变量，其中一个临时变量nextNodeTemp保存原链表的下一节点，另外一个变量prevNode保存原链表的上一节点。head用于遍历原链表，又由于需要赋予head新的next节点，所以需要先把**当前head的下一节点**保存下来(nextNodeTemp)，当赋给head的新的next节点之后，再将nextNodeTemp赋给head,这里需要赋的新的next节点就是当前head的上一个节点，所以需要利用一个变量保存下来(prevNode)的作用。



/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

public ListNode reverseList(ListNode head) {

ListNode prevNode = null;//保存上一个节点

while(head != null){

//临时变量定义在循环体内部即可

**ListNode nextNodeTemp = head.next;//临时变量：先将下一个节点保存下来**

**head.next = prevNode;//给next赋新值**

**prevNode = head;//保存当前节点**

**head = nextNodeTemp;//将下一节点在赋给head**

}

return **prevNode**;

}

}

## 算法2：值倒转

原理：将链表的数值val都顺序保存在ArrayList中，然后再倒序赋给原链表，这样就可以实现链表的逆。

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

public ListNode reverseList(ListNode head) {

//只是需要把数值倒置，link不用变

if(head==null){

return head;

}

**ArrayList<Integer> arr = new ArrayList<Integer> ();**

**ListNode temp =head;**

while (temp!=null){

arr.add(temp.val);

temp = temp.next;

}

int s= arr.size();

**temp = head;**

while (temp!=null){

temp.val=arr.get(s-1);

s--;

temp=temp.next;

}

return head;

}

}

## 算法2：自己写的

思路：一开始不知道利用临时变量那样保存下一节点，而是利用创建新的节点的方法，这种方法不太可取。

class Solution {

public ListNode reverseList(ListNode head) {

if(head == null){

return head;

}

ListNode newHead = null;

newHead = new ListNode(head.val);

newHead.next = null;

while(head.next != null){

ListNode newHead2 = new ListNode(head.next.val);//通过复制实现

newHead2.next = newHead;

newHead = newHead2;

head = head.next;

}

return newHead;

}

}

## 算法4:通过栈实现，性能很差，只是掌握Deque的使用即可

思路：将所有的ListNode存入Stack中，然后再一一获取出来。主要是考虑到栈的LIFO。

通过栈实现：

class Solution {

public ListNode reverseList(ListNode head) {

if(head == null)

return head;

Deque<ListNode> deque = new ArrayDeque<>();

//进栈

while(head != null){

deque.push(head);

head = head.next;

}

//一一出栈

ListNode preNode = deque.pop();

head = preNode;

while(preNode != null){

ListNode node = null;

if(deque.size() > 0)

node = deque.pop();

preNode.next = node;

preNode = node;

}

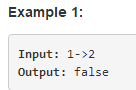
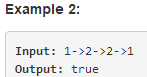
return head;

}

}

# 234. Palindrome Linked List

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

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

## 算法：将前一半LinkedList逆转(掌握)

思路：分成三步走：①获取Linked list的总长度；②将前一半Linked list元素逆转；③从中间开始判断两边是否对称(注意长度的奇偶问题)。

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode(int x) { val = x; }

\* }

\*/

class Solution {

public boolean isPalindrome(ListNode head) {

//思路：首先计算总长度；其次逆转前一半节点；最后从中间开始往两边判断。

**//first**

if(head == null) return true;

int len = 0;

ListNode temp = head;

while(temp != null){

len++;

temp = temp.next;

}

**//second:逆转前一半节点**

ListNode prevNode = null;

for(int i = 0;i < len/2;i++){

ListNode nextNodeTemp = head.next;

head.next = prevNode;

prevNode = head;

head = nextNodeTemp;

}

**//third**

// ListNode left = prevNode;

ListNode right = len%2 == 0 ? head : head.next;

while(prevNode != null && right != null){

if(prevNode.val != right.val){

return false;

}

prevNode = prevNode.next;

right = right.next;

}

return true;

}

}

## 算法2：自己的方法(不建议)

思路：将所有的值存放到ArrayList中，然后转成数组，根据索引判断两端数据是否相等。

class Solution {

public boolean isPalindrome(ListNode head) {

ArrayList<Integer> arr = new ArrayList<Integer>();

while(head != null){

arr.add(head.val);

head = head.next;

}

Integer[] vals = new Integer[arr.size()];

vals = arr.toArray(vals);

int j = vals.length-1;

for(int i = 0;i < vals.length/2;i++,j--){

if(!vals[i].equals(vals[j])){

return false;

}

}

return true;

}

}

# 225. Implement Stack using Queues

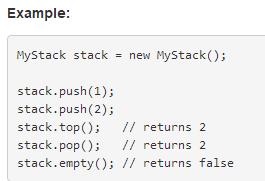
Implement the following operations of a stack using queues.

push(x) -- Push element x onto stack.

pop() -- Removes the element on top of the stack.

top() -- Get the top element.

empty() -- Return whether the stack is empty.



Notes:

You must use only standard operations of a queue -- which means only **push** to back, **peek/pop** from front, **size**, and is empty operations are valid.

Depending on your language, queue may not be supported natively. You may simulate a queue by using a **list** or **deque** (double-ended queue), as long as you use only standard operations of a queue.

You may assume that all operations are valid (for example, no pop or top operations will be called on an empty stack).

## 实现：

只能利用**Queue接口**的标准操作，也就是说那些addFirst、addLast不能使用。

class MyStack {

/\*\* Initialize your data structure here. \*/

Queue<Integer> queue;

public MyStack() {

queue = new LinkedList<>();

}

/\*\* Push element x onto stack. \*/

public void push(int x) {

queue.offer(x);

for (int i = 0; i < queue.size() - 1; i++) {

queue.offer(queue.poll());

}

}

/\*\* Removes the element on top of the stack and returns that element. \*/

public int pop() {

return queue.poll();

}

/\*\* Get the top element. \*/

public int top() {

return queue.peek();

}

/\*\* Returns whether the stack is empty. \*/

public boolean empty() {

return queue.isEmpty();

}

}

/\*\*

\* Your MyStack object will be instantiated and called as such:

\* MyStack obj = new MyStack();

\* obj.push(x);

\* int param\_2 = obj.pop();

\* int param\_3 = obj.top();

\* boolean param\_4 = obj.empty();

\*/

## 不符合要求：利用LinkedList实现

class MyStack {

**LinkedList<Integer> queue ;**

/\*\* Initialize your data structure here. \*/

public MyStack() {

queue = new LinkedList<Integer>();

}

/\*\* Push element x onto stack. \*/

public void push(int x) {

queue.addFirst(x);

}

/\*\* Removes the element on top of the stack and returns that element. \*/

public int pop() {

return queue.removeFirst();

}

/\*\* Get the top element. \*/

public int top() {

return queue.getFirst();

}

/\*\* Returns whether the stack is empty. \*/

public boolean empty() {

// return queue.size()==0?true:false;

return queue.isEmpty();

}

}

/\*\*

\* Your MyStack object will be instantiated and called as such:

\* MyStack obj = new MyStack();

\* obj.push(x);

\* int param\_2 = obj.pop();

\* int param\_3 = obj.top();

\* boolean param\_4 = obj.empty();

\*/