# 题目

给定一个链表，两两交换其中相邻的节点，并返回交换后的链表。

你不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。

**示例:**

给定 1->2->3->4, 你应该返回 2->1->4->3.

# 分析

## 方法一：递归法

**代码（推荐）：**

class Solution {

public:

ListNode\* swapPairs(ListNode\* head) {

if (head == nullptr || head->next == nullptr) {

return head;

}

ListNode\* newHead = head->next;

head->next = swapPairs(newHead->next);

newHead->next = head;

return newHead;

}

};

或：

**思路：**

1、从链表的头节点head开始递归。

2、每次递归都负责交换一对节点。由firstNode和secondNode表示要交换的两个节点。

3、下一次递归则是传递的是下一对需要交换的节点。若链表中还有节点，则继续递归。

4、交换了两个节点以后，返回secondNode，因为它是交换后的新头。

5、在所有节点交换完成以后，我们返回交换后的头，实际上是原始链表的第二个节点。

**代码：**

class Solution

{

public:

ListNode\* swapPairs(ListNode\* head)

{

///终止条件为所有都交换完了的链表

if(head == NULL || head->next == NULL)

return head;

///递归单元内两个节点做交换

//定义一个p结点为head->next(该递归单元内第二个节点)

ListNode \*p = head->next;

//记录第三个节点同时也是下一次递归的head

ListNode \*temp = p->next;

//使此单元内第二个节点指向第一个节点，即交换位置

p->next = head;

//交换完成后的head应指向下一次单元返回的头节点

head->next = swapPairs(temp);

//返回该单元的头节点

return p;

}

};

**复杂度分析：**

时间复杂度：O(N)，其中N指的是链表的节点数量。

空间复杂度：O(N)，递归过程使用的堆栈空间。

## 方法二：迭代法

思路：

1、创建一个虚拟头节点 dummy，将其 next 指针指向链表的头节点 head。

2、使用指针 prev 来标记虚拟头节点或者上一对交换节点的前一个节点，初始时指向虚拟头节点。

3、迭代地遍历链表，每次取出两个相邻节点 node1 和 node2 进行交换，同时更新 prev 指针。

4、将交换后的节点连接到 prev 节点后，并更新 prev 指针为交换后的第二个节点 node2。

5、继续迭代直到链表末尾或者只剩一个节点为止。

6、返回虚拟头节点的 next 指针，即为交换后的链表头节点。

代码：

class Solution {

public:

ListNode\* swapPairs(ListNode\* head) {

// 创建一个虚拟头节点，简化边界情况的处理

ListNode\* dummy = new ListNode(0);

dummy->next = head;

// 使用 prev 指针来标记前一个交换节点或者虚拟头节点

ListNode\* prev = dummy;

// 循环直到链表末尾或者只剩一个节点为止

while (prev->next && prev->next->next) {

// 获取需要交换的两个相邻节点

ListNode\* node1 = prev->next;

ListNode\* node2 = prev->next->next;

// 交换节点，并更新 prev 指针

prev->next = node2;

node1->next = node2->next;

node2->next = node1;

prev = node1;

}

// 返回虚拟头节点的下一个节点，即为交换后的链表头节点

return dummy->next;

}

};

或：

**思路：**

我们把链表分为两部分，即奇数节点为一部分，偶数节点为一部分，A指的是交换节点中的前面的节点，B指的是要交换节点中的后面的节点。在完成它们的交换，我们还得用prevNode记录A的前驱节点。

算法：

1、firstNode（即A）和secondNode（即B）分别遍历偶数节点和奇数节点，即两步看作一步。

2、交换两个节点：

firstNode.next = secondNode.next

secondNode.next = firstNode

3、还需要更新 prevNode.next 指向交换后的头。

prevNode.next = secondNode

4、迭代完成后得到最终的交换结果。

**代码：**

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode(int x) : val(x), next(NULL) {}

\* };

\*/

class Solution

{

public:

ListNode\* swapPairs(ListNode\* head)

{

//新建一个空结点，用来指向头节点

ListNode\* p = new ListNode(0);

p->next = head;

//新建和p相同一个curr节点，两个相同的节点一个是当前做改变的节点，一个是保持不动用来返回的节点

ListNode\* curr = p;

//循环条件为当前节点为NULL或当前的下一个节点为NULL，分别对应偶数和奇数个节点的终止标志

while(head != NULL && head->next != NULL)

{

//为了清晰明了，我们新建两个节点，一节点和二节点

ListNode\* firstNode = head;

ListNode\* secondNode = head->next;

///把一和二进行交换，并连接前后

//当前curr节点指向二节点

curr->next = secondNode;

//一节点指向二节点此时的下一节点

firstNode->next = secondNode->next;

//二节点指向一节点，即交换位置成功

secondNode->next = firstNode;

//由于每次循环curr节点都指向每次循环的一节点，所以要再次把curr节点指向一节点

curr = firstNode;

//每次移动都是由head节点来赋值操作，所以head应向右移动两格，即新循环的一节点

head = firstNode->next;

}

//返回p的下一个节点即对应整个操作后的链表

return p->next;

}

};

**另一种写法：**

class Solution {

public:

ListNode\* swapPairs(ListNode\* head) {

ListNode\* dummyHead = new ListNode(0);

dummyHead->next = head;

ListNode\* temp = dummyHead;

while (temp->next != nullptr && temp->next->next != nullptr) {

//与反转链表对比分析

ListNode\* node1 = temp->next;

ListNode\* node2 = temp->next->next;

temp->next = node2;

node1->next = node2->next;

node2->next = node1;

temp = node1;

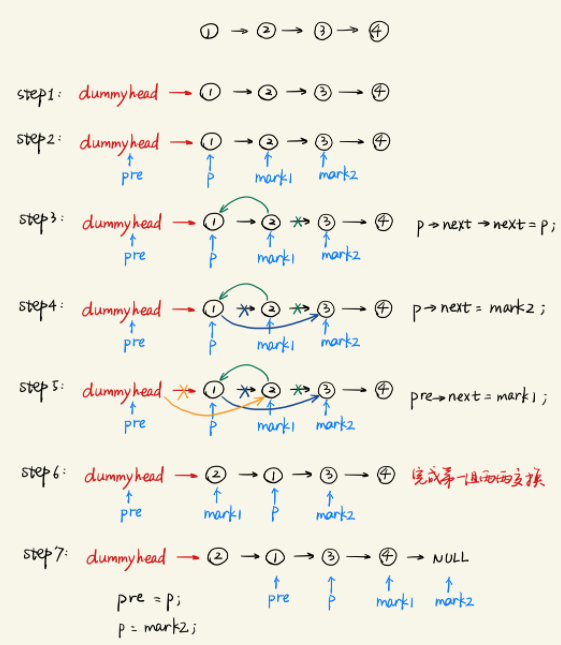
}

return dummyHead->next;

}

};

另一种写法：



/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode(int x) : val(x), next(NULL) {}

\* };

\*/

class Solution {

public:

ListNode\* swapPairs(ListNode\* head) {

if(!head || !head->next) return head;

ListNode dummyhead(-1);

dummyhead.next = head;

ListNode \*pre = &dummyhead;

ListNode \*p = head;

ListNode \*mark1, \*mark2;

while(p && p->next){

//注意循环条件，要循环到处理完全结束or只剩一个节点无需交换

mark1 = p->next;

mark2 = p->next->next;

p->next->next = p;

p->next = mark2;

pre->next = mark1;

pre = p;

p = mark2;

}

return dummyhead.next;

}

};