# 定义

为了克服数组的缺点，引入了链表。

线性表是N个数据元素的有限序列。

说明：这里的数据元素可以是简单的一个变量，比如姓名，也可以是复杂的一个结构体，比如包含id，name，age的Person结构。

**特点：**

N个节点离散分配

彼此通过指针相连

每个节点只有一个前驱节点，每个节点只有一个后续节点

首节点没有前驱节点，尾节点没有后续节点

**专业术语：**

## 首节点

**首节点（哑结点）：**第一个有效的节点

## 尾节点

**尾节点：**最后一个有效的节点

## 头结点

**头结点：**在首节点前面添加的一个节点，头结点没有存放有效数据和有效节点个数，这样主要是**方便链表增删改查统一操作（否则需要单独考虑首节点的操作）**。

**头结点数据域一般无意义**，但也可以用来存放链表的长度（主要是用头结点的指针域获取首节点地址）。

**头结点不一定是链表的必需元素**。

## 头指针

**头指针：**指向链表第一个结点（可能是头结点）的指针变量，是链表的**必要元素**

头指针具有标识作用，所以常用头指针冠以链表的名字（指针变量的名字）

无论链表是否为空，**头指针均不为空**

## 尾指针

**尾指针：**指向尾节点的指针变量

## 链表必要参数

**确定一个链表需要几个参数：**头指针（尾指针next置NULL即可）

# 分类

顺序表分类：

（1）顺序表：数组

（2）链表：静态链表，单链表，循环链表，双向链表

说明：顺序表带有下标，在遍历和寻址的时候速度快，即查找相关效率高；当插入或者删除元素的时候，需要移动数组中的元素，效率较低。

## 单链表

### 定义

单链表是单向的链表，链表的每个结点只包含一个指针域。

### 初始化

### 插入节点

### 删除节点

### 分析

## 双向链表

### 定义

双向链表的每一个节点由两个指针域和一个数据域组成，一个指针域是寻找头结点，一个是寻找尾节点。

### 初始化

// 双向链表节点定义

struct Node {

int data;

Node\* prev;

Node\* next;

Node(int val) : data(val), prev(nullptr), next(nullptr) {}

};

// 双向链表类

class DoublyLinkedList {

private:

Node\* head;

Node\* tail;

public:

// 初始化

DoublyLinkedList() : head(nullptr), tail(nullptr) {}

// 插入节点（尾插）

void insert(int val);

// 删除节点（按值删除第一个匹配项）

void remove(int val);

// 析构函数：释放内存

~DoublyLinkedList() {

Node\* current = head;

while (current) {

Node\* next = current->next;

delete current;

current = next;

}

}

};

### 插入

// 插入节点（尾插）

void insert(int val) {

Node\* newNode = new Node(val);

if (!head) { // 空链表

head = tail = newNode;

} else {

tail->next = newNode;

newNode->prev = tail;

newNode->next = nullptr;

tail = newNode;

}

}

### 删除

// 删除节点（按值删除第一个匹配项）

void remove(int val) {

Node\* current = head;

while (current) {

if (current->data == val) {

if (current->prev)

current->prev->next = current->next;

else

head = current->next; // 删除头节点

if (current->next)

current->next->prev = current->prev;

else

tail = current->prev; // 删除尾节点

delete current;

return;

}

current = current->next;

}

}

### 分析

## 循环链表

### 定义

循环链表与单链表不同的是，最后尾节点的指针域指向了第一个元素的数据域。

### 初始化

### 插入

### 删除

### 分析

## 双向循环链表

## 非循环链表

## 静态链表

### 定义

静态链表不使用链表，而是使用数组完成相关的操作。这种用数组描述链表的描述方法叫做游标实现法。

### 初始化

### 插入

### 删除

### 分析

## 松散链表

### 概述

与数组相比，链表的一项优点，就是它只需要花费O(1)级别的时间，即可在任意位置插入元素，然而要想在链表中搜索元素，则需要花费O(n)级别的时间。为了提升搜索效率，我们可以采用另一种方式实现单链表，这就是松散链表。

松散链表的每个大节点中都保存有多个元素，为了讨论方便，我们称这种大节点为块。每个这样的块，都是循环链表，用以将块内的各个小节点连接起来。

假设链表中的元素个数始终不超过n。为了简单起见，我们还假设，除了最后一块之外，前面各块中必须恰好包含√n个元素。这样的话，整个链表的块数，不会超过√n个。

### 操作

#### 查找元素

在松散链表中寻找第k个元素，需要花费O(√n)级别的时间。

#### 插入元素

#### 移动元素

### 性能

松散链表在速度与空间占用量方面都有优势。

1. 如果每一块内的元素个数设计得比较合理（例如，没有超过缓冲线的容量），那么就可以提升内存的局部性，也就是使得经常用到的这些元素在内存中能够离得比较近，这可以极大地提升缓存效果。
2. 在元素个数为n且每一块最多能保存m个元素的松散链表中，连接的数量是O(n/m)，因此可以节省很多空间。当元素本身占据的空间较小时，这种效果尤为明显。

### 对比

## 跳跃链表

### 概述

跳跃链表（skip list）是一种可以替代平衡二叉树的结构。与二叉树相比，这种链表能够迅速执行搜索、插入以及删除操作。它所达到的平衡是概率上的平衡，而不像平衡二叉树那样，是严格的平衡。这种链表，其实就是一份普通的链表，只不过其节点中还有一些指针，可以跳过链表中的某些元素而已。它采用随机数生成器来制定这些决策。

对于排好顺序的普通链表来说，其搜索、插入以及删除操作都需要耗费O(n)级别的时间，因为要把这些节点逐个扫描一遍才能找到相关的节点。如果能设法以较大的步伐来迈进，那么就可以减少扫描的开销，这正是跳跃链表的基本思路。

### 性能

如果插入的是随机数据，那么对于普通的二叉排序树来说，其搜索、插入以及删除效率比较高，可以达到O(logn)级别；但是如果输入的是排过序的数据，那么其效率则降为O(n)。然而对于跳跃链表而言，无论输入的是什么数据，这些操作的执行效率差不多都与随机构建的二叉树一样好，也就是都能达到O(logn)级别。

### 对比

简单地说，跳跃链表与排过序的普通链表相比，有两个区别：

1. 普通链表中的节点只有一个指针，也就是指向下一个节点的next指针。而跳跃链表中的节点可能包含多个指针，用以指向不同的节点，这些指针也成为提前引用（forward reference）。
2. 每个节点究竟拥有多少个提前是通过概率来决定的。

跳跃链表是分层的，每一层都有对应的提前引用。节点的层数也称为节点的大小。在普通的链表上面执行插入、删除以及查找操作，可能要把整个链表从头到尾扫描一遍，这需要花费O(n)级别的时间，然而在跳跃链表上面执行操作，只需花费O(logn)级别的时间，因为在遍历过程中，可以通过指针跳过其中的某些节点。

# 特点

链表的优缺点如下：

## 优点

**优点：**能够在常数时间内增长，而数组不能。

## 缺点

**缺点：**

1、访问单个元素花费的时间比较长；

2、在空间局部性上不如数组（分配给数组的内存是连续的）；

3、存储于获取数据时开销很大；

4、在某些情况下很难维护。

注：可以理解为数组适用于读操作多，链表适用于写操作多的场景。

# 存储

## 顺序存储

### 定义

#define MAX\_SIZE 128

typedef struct SqList{

int data[MAX\_SIZE]; //MAX\_SIZE是线性表的最大长度（数据域）

int length; //线性表需要一个变量标识当前长度（指针域）

}SqList;

注：**指针域length是必须的，防止出现越界访问**，链式存储中不需要设置，next指针指向的元素都是动态分配的，除非内存耗尽否则不存在越界访问。

### 初始化

SqList \*s; //访问成员变量s->data

SqList s; //访问成员变量s.data

### 获取元素

思路：将data数组的第i-1个字符赋值给指针型形参（注意必须是指针型的形参或引用），这样做可以获取输出

代码：

int SqListGetElement(SqList \*s,int i,int \*element){

if(s->length == 0 || i<1 || i>s->length)//对输入的参数进行检查

return -1;

\*element = s->data[i-1];

return 0;

}

注：也可以定义函数返回值类型为int\*，这样入参可以省去element。

### 插入元素

思路：先将元素依次向后移动1个位置，然后将待插入的数据放到数组的特定位置

代码：

int SqListInsertElement(SqList \*s,int i, int \*element){

int j=0;

if(s->length==MAX\_SIZE || i<1 || i>s->length+1)//线性表满了或i不在范围内

return -1;

if(i<s->length){ //插入位置不在表尾

for(j=i;j<s->length;j++){

s->data[j+1]=s->data[j]; //这种方式不对，会发生覆盖

}

}

改写为：

if(i<s->length){

**for(j=s->length-1;j>=i-1;j--){**

s->data[j+1]=s->data[j];

}

}

s->data[i] = \*element;

s->length++;

return 0;

}

### 删除元素

思路：先将指定元素删除，然后数组中元素依次向前移动1位

代码：

int SqListDeleteElement(SqList \*s, int i, int \*element){

int k;

if(s->length==0 || i<1 || i>s->length){

return -1;

}

\*element = s->data[i-1];

if(i<s->length)

{

for(k=i;k<s->length;k++){

**s->data[k-1]=s->data[k];** //向前移动

}

}

s->length--;

return 0;

}

### 分析

#### 时间复杂度

最好的情况，插入和删除操作刚好要求在最后一个位置操作，因为不需要移动任何元素，所以此时的时间复杂度为O(1)。

最坏的情况，如果要插入和删除的位置是第一个元素，那就意味着要移动所有的元素向后或者向前，所以这个时间复杂度为O(n)。

至于平均情况，就取中间值O((n-1)/2)，平均情况复杂度简化后是O(n)。

综上，线性表的顺序存储结构，在存、读数据时，不管是哪个位置，时间复杂度都是O(1)。而在插入或删除时，时间复杂度都是O(n)。

这说明，它比较适合元素个数比较稳定，不经常插入和删除元素，而更多的操作是存取数据的应用。

#### 空间复杂度

#### 优缺点

优点：

1、无需为表示表中元素之间的逻辑关系而增加额外的存储空间；

2、可以快速地存取表中任意位置的元素。

缺点：

1、插入和删除操作需要移动大量元素；

2、当线性表长度变化较大时，难以确定存储空间的容量；

3、容易造成存储空间的“碎片”（因为是一整块的申请，那么就容易造成存储块之间小块的内存被浪费）。

## 链式存储

### 定义

思路：链式存储即先定义一个独立的节点node（结点Node：数据域和指针域组成数据元素称为存储映像/结点），然后定义链表将节点串联起来，这就需要用到next指针。

代码：

typedef struct LinkListNode{

int data; //只表示一个元素，所以这里定义一个int而不是数组（数据域）

struct LinkListNode \*next;//存储下一个节点地址（指针域）

}LinkListNode; //结点（节点名称）

typedef struct LinkListNode \*LinkList; //指针型变量（**头指针**）

注：定义指针变量习惯\*放在右边，这是为了防止对于int\* i,j这种情况混淆。

或者：

typedef struct LinkListNode{

int data;

struct LinkListNode \*next;

}LinkListNode, \*LinkList; //这种定义方式比较好（推荐使用！）

或者：

typedef struct LinkListNode{

int data;

struct LinkListNode \*next;

}LinkListNode, \*LinkListPtr;

typedef struct LinkList{

LinkListPtr p;

int count;//这样定义不太合理，链式存储就没有必要设置当前长度length

}

### 初始化

LinkList \*l;

LinkList l;

### 创建链表

LinkList GenerateLinkList(int len)

LinkListPtr pHead = (LinkListPtr)malloc(sizeof(LinkListNode));

//申请内存，设置头指针

pHead ->next = NULL;

//生成链表中的各个节点

for(int i=0;i<len;i++)

{

LinkListPtr pNewNode = (LinkListPtr)malloc(sizeof(LinkListNode));

pNewNode->data = values;

pHead ->next = pNewNode;

pNewNode->next = NULL; //尾节点设置为NULL

}

注：采用这种方式其实是将新生成的节点pNewNode都挂载到pHead节点后面，而不是我们想象的最后一个节点后面。则可以定义一个指针，它永远指向最后一个节点，如下：

//头指针

LinkListPtr pHead = (LinkListPtr)malloc(sizeof(LinkListNode));

LinkListPtr pTail = pHead;

pTail->next = NULL; //必须清空指针域

//生成链表中的各个节点

for(int i=0;i<len;i++)

{

LinkListPtr pNewNode = (LinkListPtr)malloc(sizeof(LinkListNode));

pNewNode->data = values;

pTail->next = pNewNode;

pNewNode->next = NULL; //尾节点设置为NULL

pTail = pNewNode; //移动尾指针

}

### 整表创建

### 链表判空

LinkList p = pHead->next;

if(NULL == p)

{

trturn true;

}

### 是否有环

双指针

### 获取元素

**思路：**获取头指针，然后依次遍历链表，找到对应的第i个节点后返回

**代码：**

int LinkListGetElement(LinkList l, int i, int \*element){

//传引用而非指针LinkList \*l

int j=0;

LinkList p;//直接操作链表有危险，这里采用定义一个新的局部变量操作

j=1;

while(p && j<i){//依次遍历，更新指针位置

p=p->next;

j++;

}//掌握这种链表遍历的方法

if(!p||j>=i){

return -1;

}

\*element = p->data;

return 0;

}

分析：单链表的读取算法时间复杂度取决于i的位置，当i=1时，则不需要遍历，而i=n时则遍历n-1次才可以。因此最坏情况的时间复杂度为O(n)。

由于单链表中没有定义表长，所以不能实际知道要循环多少次，因此也就不方便使用for控制循环。

其核心思想叫做“工作指针后移”，这是很常用的技术。

### 链表长度

int ListLength(LinkList l)

{

LinkList tmp = l->next;

int len = 0;

while(NULL != tmp)

{

++len;

tmp = tmp->next;

}

return len;

}

### 插入元素

**思路：**插入需要新建一个节点（malloc分配），然后更新这个新节点的成员变量，即数据data和指针next，接着就是以前的节点与新节点建立联系，这就形成了一个新的链表。

**注意：**需要暂时保存next指针地址，防止被覆盖！

**代码：**

int LinkListInsertElement(LinkList l, int i, int \*element){

//这里定义LinkList l是指针类型，因前面已经定义为LinkListNode \*LinkList

//如果这里定义为LinkList \*l，则前面的定义就是LinkListNode LinkList

int j=0;

LinkList p,s;

//需要借助中间变量链表实现操作，

//1个用于接收形参链表，1个用于malloc新节点

j=1;

p=l->next; //头结点的下一个指针（next地址需要暂时保存！）

for(p && j<i){

p=p->next;

j++;

}

s=(LinkList)malloc(sizeof(LinkListNode));

//链表插入元素就需要新分配一个节点内存空间，然后更新指针的指向

s->data=\*element;

s->next=p->next;//更新这个新节点的成员变量

p->next=s;//建立节点的关联

return 0;

}

注：对于p后新增结点q的添加，可以采用如下方法：

1、使用临时变量

r = p->next;

p->next = q;

q->next = r;

2、不使用临时变量

q->next = p->next;

p->next = q;

### 删除元素

**思路：**暂时保存指针，放置删除节点后，next后续出现断链。

**代码：**

int LinkListDeleteElement(LinkList l, int i, int \*element){

int j=0;

LinkList p,q;//插入和删除都需要借助中间变量的一个节点完成

j=1;

p=l->next;

while(p && j<i){

p=p->next;

j++;

}//确定删除节点的位置

**q**=p->next;//待删除的节点（需要暂存该节点，否则出现断链）

\*element=q->data;//待删除节点的形参

p->next=q->next;//待删除节点的指针

free(q);//释放掉这个节点，删除的就不需要了

return 0；

}

分析：单链表的插入和删除时间复杂度都是O(n)。

注：以下几种错误需要格外注意

1、删除p所指结点后面结点：p->next = p->next->next;

错误，这样p->next会内存泄露，并没有完全释放

2、delete p->next;/free p->next;

错误，链表被破坏，断链了；该节点后面的结点都找不到了

应该采用临时变量：即先临时定义一个指向p后面结点的指针r

r = p->next; //r指向p后面的那个结点

p->next = r->next; //更新下一个结点

delete r;/free r; //这样p->next已经处理好，不会断链

关于链表插入和删除的注意事项：

1、特殊处理链表为空，或者链表长度为1的情况；

2、注意插入操作的调整过程；

3、注意删除操作的调整过程；

注意点：头尾节点及空节点需要特殊考虑。

双链表的插入与删除和单链表类似，但是需要额外考虑previous指针的指向。

### 整表删除

### 清空

### 销毁

void DestroyLinkList(LinkList \*list)

{

LinkList p,q;

p = \*list;

while(p){

q = p->next;

free(p);

p = q;

}

\*list = NULL;

}

注：最后需要将\*list的内容置为NULL，这样主函数中的链表就为空，放置list称为野指针。

### 分析

单链表读取、插入和删除时间复杂度都是O(n)。

如果在我们不知道第i个元素的指针位置，单链表数据结构在插入和删除操作上，与线性表的顺序存储结构是没有太大优势的。

但是，如果我们希望从第i个位置开始，插入连续10个元素，对于顺序存储结构意味着，每一次插入都需要移动n-i个位置，所以每次都是O(n)。

而对于单链表，我们只需要在第一次时，找到第i个位置的指针，此时为O(n)，接下来只是简单地通过赋值移动指针而已，时间复杂度都是O(1)。

显然，对于插入和删除数据越频繁的操作，单链表的效率优势就越明显。

# 应用

链表的基本应用包括：遍历（判断是否有环，旋转），翻转（单链表翻转，区间翻转），逆序，复制，删除（删除K个，删除重复），排序，合并，拆分。

链表问题代码实现注意事项：

1、链表调整函数的返回值类型，根据要求往往返回值是节点类型；

2、处理链表过程中，先采用画图的方式理清逻辑（防止出现断链）；

3、链表问题对于边界条件讨论要求严格。

关于链表插入和删除的注意事项：

1、特殊处理链表为空，或者链表长度为1的情况；

2、注意插入操作的调整过程；

3、注意删除操作的调整过程。

**注意点：**头尾节点及空节点需要特殊考虑。

双链表的插入与删除和单链表类似，但是需要额外考虑previous指针的指向。

1、大量链表问题可以使用额外数据结构来简化调整过程；

2、但是**链表问题最优解往往是不使用额外数据结构的方法**。

## 链表遍历

代码：

#include <iostream>

using namespace std;

typedef struct list\_node ListNode;

struct list\_node

{

struct list\_node\* next;

int value;

};

/\*

初始化List 将从1~n的数字插入到链表中

\*/

void Init\_List(ListNode\*& head,int\* array,int n)

{

head = NULL;

ListNode\* tmp;

ListNode\* record;

for(int i=1;i<=n;i++)

{

tmp = new ListNode;

tmp->next = NULL;

tmp->value = array[i-1];

if(head == NULL)

{

head = tmp;

record = head;

}

else

{

record->next = tmp;

record = tmp;

}

}

}

void print\_list(ListNode\* list)

{

ListNode\* tmp = list;

while(tmp != NULL)

{

cout<<tmp->value<<endl;

tmp = tmp->next;

}

}

### 求单链表节点个数

//求单链表中节点的个数

int GetListLength(ListNode\* phead)

{

if(phead == NULL)

return 0;

int length = 0;

ListNode\* current = phead;

while(current != NULL)

{

length++;

current = current->next;

}

return length;

}

如何判断链表的总结点数是奇数还是偶数？

采用每次移动两个节点的指针来判断。如果该指针最后指向的是NULL，那么说明总节点数为偶数；若指向链表的最后一个节点，则说明总节点数为奇数。

class Solution {

public:

    int IsLinkedListLengthEven(ListNode\* head) {

        while(head && head->next)

        {

            head = head->next->next;

        }

        if(!head)

            return 0;

        return 1;

    }

};

### 递归方式求链表长度

//递归的方法求解链表的长度

int Len\_list(ListNode\* list)

{

if(list == NULL)

return 0;

else

return Len\_list(list->next)+1;

}

### 环形链表/判断单链表中是否有环

**题目一：**已知链表中可能存在环，若有环返回环起始节点，否则返回NULL。



注：Leetcode 141

**题目二：**为了表示给定链表的环，我们使用整数pos来表示链表尾连接到链表中的位置（索引从0开始），结果pos是-1，则在该链表中没有环。

**分析：**

思路1：使用set求环起始节点

1. 遍历链表，将链表中节点对应的指针（地址），插入set；
2. 在遍历时插入节点前，需要在set中查找，第一个在set中发现的节点地址，即是链表环的起点。



思路2：使用快慢指针（双指针），如果链表中有环，则一定会相遇。















**代码：**

#### 方法一：set



#### 方法二：双指针



另外一种：

//判断单链表中是否有环

bool HasCircle(ListNode\* phead)

{

ListNode\* pfast = phead;//快指针每次前进两步

ListNode\* pslow = phead;//慢指针每次前进一步

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

{

pfast = pfast->next->next;

pslow = pslow->next;

if(pslow == pfast)

return true;

}

return false;

}

**测试：**



### 旋转链表

注：Leetcode 61

### [链表的中间结点](https://leetcode-cn.com/problems/middle-of-the-linked-list)

### 链表的倒数第k个节点

### [回文链表](https://leetcode-cn.com/problems/palindrome-linked-list)

### 寻找节点

1、寻找与模除有关的节点

编写函数，以便根据整数常量k，在包含n个元素的单链表中，寻找序号最大，且该序号又能为k所整除的那个节点。如果n=19,且k=3，那么就返回第18个节点。

代码：

### 两链表交点

**题目：**已知链表A的头节点指针headA，链表B的头节点指针headB，两个链表相交，求两链表交点对应的节点。



注：Leetcode 160



1、如果两个链表没有交点，则返回NULL；

2、在求交点的过程中，不可以破坏链表的结构或者修改链表的数据域；

3、可以确保传入的链表A与链表B没有任何环；

4、实现算法尽可能使时间复杂度O(n)，空间复杂度O(1)

**思路：**

**方法一：**

可以利用STL的set实现：



1、遍历链表A，将A中节点对应的指针（地址），插入set；

2、遍历链表B，将B中节点对应的指针（地址），在set中查找，发现在set中的第一个节点地址，即是两个链表的交点。



注：存入set的是地址而不是值域，我们判断的是交点，交点即同一个地址上的数据。

分析：时间复杂度O(nlogn)，空间复杂度O(n)。

**方法二：**

方法一比较简单，但是并不是最优的，可以使用基于数学的方法，实现空间复杂度O(1)。

步骤1：计算headA链表长度，计算headB链表长度，较长的链表多出的长度



步骤2：将较长的链表的指针移动到和较短链表指针对齐的位置：



步骤3：headA与headB同时移动，当两指针指向同一个节点时，即找到了



**代码：**

**方法一：**利用set集合特性求交集



**方法二：**



**测试：**



## 链表节点查找

### 带环的单链表中入环的第一个节点

题目要求：在一个存在环的单链表中，查找进入环的第一个节点。

代码：

//已知一个单链表中存在环，求进入环中的第一个节点

ListNode\* GetFirstNodeInCircle(ListNode\* phead)

{

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

{

return NULL;

}

ListNode\* pfast = phead;

ListNode\* pslow = phead;

//先判断是否存在环

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

{

pslow = pslow->next;

pfast = pfast->next->next;

if(pslow == pfast)

break;

}

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

return NULL;

//如果存在环，快慢指针都以同样的速度前进，相遇即为第一个节点

pfast = phead;

while(pslow != pfast)

{

pslow= pslow->next;

pfast = pfast->next;

}

return pslow;

}

//按照一定的要求删除链表中的节点

typedef boolk (\*remove\_fn)(ListNode\* node);

ListNode\* remove\_if(ListNode\* head,remove\_fn rm)

{

for(ListNode\* prev= NULL,\*curr = head;curr != NULL)

{

ListNode\* next = curr->next;

if(rm(curr))

{

if(prev)

prev->next = next;

else

head= next;

delete curr;

}

else

prev= curr;

curr = next;

}

return head;

}

// 升级版(使用二级指针删除单链表中的节点)

void remove\_if(ListNode\*\* head,remove\_if rm)

{

for(ListNode\*\* curr = head;\*curr;)

{

ListNode\* entry = \*curr;

if(rm(entry))

{

\*curr = entry->next;

delete entry;

}

else

curr = &entry->next;

}

} //仔细体味！

## 链表翻转

### 链表逆序（头插法）



**代码（两种方法对比）：**





### 反转整个链表

#### 翻转单链表

//翻转单链表

ListNode\* ReverseList(ListNode\* phead)

{

//如果单链表为空或者只有一个节点，无需翻转，直接返回头节点

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

return phead;

ListNode\* preverse = NULL;

//翻转后的新链表头指针，初始化为NULL

ListNode\* current = phead;

while(current != NULL)

{

ListNode\* temp = current;

current = current->next;

temp->next = preverse;

//将当前节点记录，插入新链表的最前端

preverse = temp;

}

return preverse;

}

int main()

{

int array[]={1,2,3,4,5,6,7,8,9,10};

ListNode\* list;

Init\_List(list,array,sizeof(array)/sizeof(int));

print\_list(list);

list = ReverseList(list);

print\_list(list);

return 0;

}

拓展：单链表和双链表的翻转（参考MariaDB的源码）。

#### 整个链表逆序

**题目：**已知链表头节点指针head，将链表逆序（不可申请额外空间）。

注：Leetcode 206（该种方法很重要！）



**思路：**依次遍历链表节点，每遍历一个节点即逆置一个节点



循环1次：



循环2次：



循环3次：



循环4次：



循环5次：



例如如果需要对如下场景做逆序：



 需要分别执行以下操作：

注：因为需要修改原始链表head的next指针，所以需要暂时保存next指针。

**代码：**



注：代码实际上就是移动head和new\_head的指针。

**测试：**



#### 反转链表

注：LeetCode 206

### 反转部分链表

#### 单链表中区间节点的翻转/翻转链表II

题目要求：在一个单链表中，给定两个值m和n，翻转该链表中m到n这段区间的节点。

例如给定一个单链表：1->2->3->4->5->NULL

如果给定值m=2andn=4，操作之后返回的链表为1->4->3->2->5->NULL

另一种描述：

给你单链表的头指针 head 和两个整数 left 和 right ，其中 left <= right 。请你反转从位置 left 到位置 right 的链表节点，返回 反转后的链表 。

注：LeetCode 92

代码：

#include <iostream>

#include <vector>

#include <string>

using namespace std;

typedef struct list\_node List;

struct list\_node

{

int value;

struct list\_node\* next;

};

void Init\_List(List\*& head,int\* array,int n)

{

head = NULL;

List\* tmp;

List\* record;

for(int i=1;i<=n;i++)

{

tmp = new List;

tmp->next = NULL;

tmp->value = array[i-1];

if(head == NULL)

{

head = tmp;

record = head;

}

else

{

record->next = tmp;

record = tmp;

}

}

}

void print\_list(List\* list)

{

List\* tmp=list;

while(tmp != NULL)

{

cout<<tmp->value<<endl;

tmp = tmp->next;

}

}

/\*

将链表中的第m个节点到第n个节点之间的元素进行翻转

\*/

void ReverseList(List\*& list,int m,int n)

{

if(list == NULL ||list->next == NULL || n-m<1)

return ;

int num =1;

List\* pre,\*next,\*cur,\*temp,\*tmp;

cur = list;

pre = NULL;

while(cur != NULL)

{

next = cur->next;

if(num < m)

{

pre = cur;

cur = next;

}

if(num == m)

{

tmp = cur;

temp = cur;

cur = next;

}

if(num >m && num <= n)

{

cur->next = temp;

temp = cur;

cur = next;

}

if(num == n)

{

if(m ==1)

list = temp;

else

pre->next = temp;

tmp->next = cur;

break;

}

num++;

}

}

int main()

{

int array[]={5,1,2,7,8,4,3,6,10,9};

List\* list ;

Init\_List(list,array,sizeof(array)/sizeof(int));

ReverseList(list,1,3);

print\_list(list);

return 0;

}

#### 指定位置逆序

**题目：**已知链表头结点指针head，将链表从位置m到n逆序（不申请额外空间）。



注：Leetcode 92

**思路：**



逆置段头结点（2）的前驱（1）：逆置后该前驱节点不再指向逆置段头结点（2），而是指向逆置段尾节点（4）。

逆置前头结点/逆置后尾节点（2）：从哪里开始逆置，逆置后指向逆置段尾节点后继（5）。

逆置前尾节点/逆置后头结点（4）：需要将逆置前头结点前驱（1）指向该节点。

逆置段尾节点的后继（5）：作为逆置后尾节点（2）的后继。

步骤1：将head向前移动m-1（1）个位置（从1移动到2），找到开始逆置的节点，记录该节点前驱（pre\_head）、该节点（head），将逆序前头结点head存储在modify\_list\_tail中，逆置后指向后继5：



步骤2：从head节点开始，逆置节点



步骤3：将pre\_head与new\_head连接，modify\_list\_tail与head连接



**思考：**

1. 最终结果应该返回哪个节点？
2. 如果m=1，有什么特殊的？

**代码：**



**测试：**



#### 反转链表II

反转从位置 m 到 n 的链表。请使用一趟扫描完成反转。

**说明:**

1 ≤m≤n≤ 链表长度。

注：LeetCode 92

分析：

### 分组反转

#### 单链表的K翻转

题目要求：在一个单链表中，给定一个值K，使得每K个节点都进行一次翻转。

例如给定一个单链表：1🡪2🡪3🡪4🡪5

如果给定K=2，操作之后返回的链表为2🡪1🡪4🡪3🡪5

如果给定K=3，操作之后返回的链表为3🡪2🡪1🡪4🡪5

代码：

#include <iostream>

#include <vector>

using namespace std;

/\*

按照要求做K翻转

\*/

typedef struct list\_node List;

struct list\_node

{

int value;

struct list\_node\* next;

};

void Init\_List(List\*& head,int\* array,int n)

{

head = NULL;

List\* tmp;

List\* record;

for(int i=1;i<=n;i++)

{

tmp = new List;

tmp->next = NULL;

tmp->value = array[i-1];

if(head == NULL)

{

head = tmp;

record = head;

}

else

{

record->next = tmp;

record = tmp;

}

}

}

void print\_list(List\* list)

{

List\* tmp=list;

while(tmp != NULL)

{

cout<<tmp->value<<endl;

tmp = tmp->next;

}

}

/\*

翻转链表List 使得新的头部为Head，新的尾部为tail

\*/

void Reverse\_list(List\*& list,List\*& head,List\*& tail)

{

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

return;

head = list;

tail = list;

List\* cur = NULL;

List\* next;

while(head != NULL)

{

next = head->next;

head->next = cur;

cur = head;

head = next;

}

list= cur;

head = cur;

}

/\*

做K个节点的翻转

\*/

void Reverse\_k(List\*& list,int k)

{

int num =1;

int flag =1;

if(list == NULL ||list->next == NULL || k ==0)

return ;

List\* head,\*tail,\*next,\*pre;

head = list;

tail= list;

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

{

tail = tail->next;

num++;

if(num == k)

{

if(tail != NULL)

{

next = tail->next;

tail->next = NULL;

}

else

next = NULL;

Reverse\_list(head,head,tail);

if(flag)

{

list = head;

flag =0;

pre = tail;

}

else//第二次之后的翻转

{

pre->next = head;

pre = tail;

}

head = next;

tail = next;

num =1;

}

}

pre->next = head;

}

List\* Reverse(List\* pre,List\* end)

{

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

return pre;

List\* head = pre->next;

List\* cur = pre->next->next;

while(cur != end)

{

List\* next = cur->next;

cur->next = pre->next;

pre->next = cur;

cur = next;

}

head->next = end;

return head;

}

//另一种比较简单的方法

List\* Reverse\_K(List\* head,int k)

{

if(head == NULL)

return NULL;

List\* dummy = new List;

dummy->next = head;

int count =0;

List\* pre = dummy;

List\* cur = head;

while(cur != NULL)

{

count++;

List\* next = cur->next;

if(count == k)

{

pre = Reverse(pre,next);

count =0;

}

cur = next;

}

return dummy->next;

}

int main()

{

int array[]={1,2,3,4,5,6,7,8,9,10,11};

List\* list,\*head,\*tail;

Init\_List(list,array,sizeof(array)/sizeof(int));

list = Reverse\_K(list,3);

print\_list(list);

return 0;

}

#### K 个一组翻转链表

给你一个链表，每k个节点一组进行翻转，请你返回翻转后的链表。

k是一个正整数，它的值小于或等于链表的长度。

如果节点总数不是k的整数倍，那么请将最后剩余的节点保持原有顺序。

进阶：

你可以设计一个只使用常数额外空间的算法来解决此问题吗？

你不能只是单纯的改变节点内部的值，而是需要实际进行节点交换。

注：LeetCode 25

### 旋转链表

给你一个链表的头节点 head ，旋转链表，将链表每个节点向右移动 k 个位置。

注：LeetCode 61

解题思路：这种需要移动K位置的可以采用快慢指针。

## 链表逆序

### 两两交换链表节点

给你一个链表，两两交换其中相邻的节点，并返回交换后链表的头节点。你必须在不修改节点内部的值的情况下完成本题（即，只能进行节点交换）。

注：Leetcode 24

解题思路：采用迭代的方法模拟。

## 单链表排序

### 单链表的排序

题目要求：

代码：

#include <iostream>

#include <vector>

using namespace std;

/\*

对于链表的排序 使用归并排序最好

\*/

typedef struct list\_node List;

struct list\_node

{

struct list\_node\* next;

int value;

};

void print\_list(List\* list)

{

List\* tmp=list;

while(tmp != NULL)

{

cout<<tmp->value<<endl;

tmp = tmp->next;

}

}

/\*

初始化List 将从1~n的数字插入到链表中

\*/

void Init\_List(List\*& head,int\* array,int n)

{

head = NULL;

List\* tmp;

List\* record;

for(int i=1;i<=n;i++)

{

tmp = new List;

tmp->next = NULL;

tmp->value = array[i-1];

if(head == NULL)

{

head = tmp;

record = head;

}

else

{

record->next = tmp;

record = tmp;

}

}

}

//求链表长度

int Len\_list(List\* list)

{

if(list == NULL)

return 0;

else

return Len\_list(list->next)+1;

}

void FindMid(List\*& list,List\*& pre,List\*& last)

{

pre = list;

last = list->next;

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

{

pre = pre->next;

last = last->next;

if(last->next != NULL)

last = last->next;

}

last = pre->next;

pre->next = NULL;

pre = list;

}

//合并两个有序的链表

void Merge(List\*& list,List\*& pre,List\*& last)

{

if(pre == NULL)

{

list = last;

return;

}

if(last == NULL)

{

list =pre;

return;

}

List\* cur;

List\* tmp;

if(pre->value > last->value)

swap(pre,last);

//始终将Last链表中的节点插入到pre链表中

list = pre;

cur = pre;

while(cur->next != NULL && last != NULL)

{

if(cur->next->value > last->value)//插入元素

{

tmp = last->next;

last->next = cur->next;

cur->next = last;

cur = last;

last = tmp;

}

else

cur =cur->next;

}

if(last != NULL)

cur->next = last;

}

//更好一点的合并

void Merge\_sec(List\*& list,List\*& pre,List\*& last)

{

List\* tmp = new List;

list = tmp;

while(pre != NULL && last != NULL)

{

if(pre->value < last->value)

{

tmp->next = pre;

pre = pre->next;

}

else

{

tmp->next = last;

last = last->next;

}

tmp = tmp->next;

}

if(last != NULL)

tmp->next = last;

else

tmp->next = pre;

list = list->next;

}

/\*

归并排序的主体思想

\*/

void MergeSort(List\*& list)

{

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

return ;

//找到中间点

List\* pre = NULL;

List\* last = NULL;

FindMid(list,pre,last);//将一个链表List从中间分成pre和last两部分

MergeSort(pre);//归并排序使得前半部分有序

MergeSort(last);//归并排序使得后半部分有序

Merge(list,pre,last);//将量部分的链表合并

}

/\*

归并排序第二种实现

\*/

List\* Merge(List\* first,List\* second)

{

List\* head = NULL;

List\* current = NULL;

if(first == NULL)

return second;

if(second == NULL)

return first;

if(first->value > second->value)

{

current = first;

first = second;

second = current;

}

head = first;

current = first;

first = first->next;

//始终将second的节点插入到first链表中

while(first != NULL && second != NULL)

{

List\* temp = NULL;

if(first->value > second->value)

{

temp = second->next;

current->next = second;

second->next = first;

current = second;

second = temp;

}

else

{

current = first;

first = first->next;

}

}

if(first == NULL)

current->next = second;

return head;

}

List\* MergeSort(List\* list,int size)

{

if(size == 0 | size == 1)

return list;

//找中间点

List\* middle = list;

int i;

for(i =1;i<size/2;i++)

middle = middle->next;

List\* temp = middle->next;

middle->next =NULL;

middle = temp;

List\* left = MergeSort(list,i);

List\* right = MergeSort(middle,size-i);

return Merge(right,left);

}

int main()

{

int array[]={7,4,9,15,2,1,6,10,12,11};

List\* head;

Init\_List(head,array,sizeof(array)/sizeof(int));

head = MergeSort(head,10);

print\_list(head);

return 0;

}

### 排序链表

给你链表的头结点 head ，请将其按 升序 排列并返回 排序后的链表 。

注：LeetCode 148

分析：可以采用快慢指针将链表分割为两个子链表，然后排序，最后merge。

### 对链表进行插入排序

给定单个链表的头 head ，使用 插入排序 对链表进行排序，并返回 排序后链表的头 。

插入排序算法的步骤：

1、插入排序是迭代的，每次只移动一个元素，直到所有元素可以形成一个有序的输出列表。

2、每次迭代中，插入排序只从输入数据中移除一个待排序的元素，找到它在序列中适当的位置，并将其插入。

3、重复直到所有输入数据插入完为止。

下面是插入排序算法的一个图形示例。部分排序的列表(黑色)最初只包含列表中的第一个元素。每次迭代时，从输入数据中删除一个元素(红色)，并就地插入已排序的列表中。

注：LeetCode 147

分析：

### 重排链表/单链表节点的交叉重排

题目要求：给定一个单链表如下所示：L0->L1->……->Ln-1->Ln，重新排序使其变成L0->Ln->L1->Ln-1->L2->Ln-2->……

注：

LeetCode143

代码：

#include <iostream>

#include <vector>

using namespace std;

/\*

按照要求重新为单链表排序

\*/

typedef struct list\_node List;

struct list\_node

{

struct list\_node\* next;

int value;

};

void print\_list(List\* list)

{

List\* tmp=list;

while(tmp != NULL)

{

cout<<tmp->value<<endl;

tmp = tmp->next;

}

}

/\*

初始化List 将从1~n的数字插入到链表中

\*/

void Init\_List(List\*& head,int\* array,int n)

{

head = NULL;

List\* tmp;

List\* record;

for(int i=1;i<=n;i++)

{

tmp = new List;

tmp->next = NULL;

tmp->value = array[i-1];

if(head == NULL)

{

head = tmp;

record = head;

}

else

{

record->next = tmp;

record = tmp;

}

}

}

//求链表长度

int Len\_list(List\* list)

{

if(list == NULL)

return 0;

else

return Len\_list(list->next)+1;

}

/\*

链表的翻转

\*/

void Reverse(List\*& list)

{

List\* tmp = NULL;

List\* cur = list;

List\* next = list->next;

while(next != NULL)

{

cur->next = tmp;

tmp = cur;

cur = next;

next = next->next;

}

cur->next = tmp;

list = cur;

}

/\*

重新排序链表，将一个链表拆分，然后重新组合

关键点在于链表个数是偶数还是奇数

\*/

void Reorder\_list(List\*& list)

{

List\* first = list;

List\* second;

List\* tmp\_first,\*tmp\_second;

//需要根据链表中节点的个数来分割链表

int len = Len\_list(first);

int i;

if(len%2 == 0)

{

for(i=1;i<len/2;i++)

first = first->next;

}

else

{

for(i=1;i<len/2+1;i++)

first = first->next;

}

second = first->next;

first->next = NULL;

//将后面的链表进行翻转

Reverse(second);

//重新规划链表

first = list;

//开始进行合并，同时second链表的个数肯定不会比first链表的节点数多

while(second != NULL)

{

tmp\_first = first->next;

tmp\_second = second->next;

first->next= second;

second->next = tmp\_first;

second = tmp\_second;

first = tmp\_first;

} //能否使用伪指针将两个链表串联

}

int main()

{

int array[]={1,2,3,4,5,6,7,8,9,10,11};

List\* head;

Init\_List(head,array,sizeof(array)/sizeof(int));

Reorder\_list(head);

print\_list(head);

return 0;

}

### [分隔链表](https://leetcode-cn.com/problems/partition-list)

题目：

给你一个头结点为 head 的单链表和一个整数 k ，请你设计一个算法将链表分隔为 k 个连续的部分。

每部分的长度应该尽可能的相等：任意两部分的长度差距不能超过 1 。这可能会导致有些部分为 null 。

这 k 个部分应该按照在链表中出现的顺序排列，并且排在前面的部分的长度应该大于或等于排在后面的长度。

返回一个由上述 k 部分组成的数组。

注：LeetCode 725

分析：

可以采用如下两种思路：

1、快慢指针

2、两个链表模拟（对于这类分割问题都可以采用这种方式，比如单链表分割，重排链表等）

### 单链表的分割

题目要求：给定一个单链表L和一个值x，经过处理，使得小于x的节点值在前面，不小于x的节点值在后面。

例如：给定一个链表1->4->3->2->5-> 2和一个值x=3，链表经过处理之后变成1->2->2->4->3->5

分析：

1、简单做法

1）将链表的所有节点放入到数组中，然后将数组进行快排划分的调整过程；

2）然后将数组中的节点依次重新串连。

2、最优解



将结果分成小于/等于/大于x的链表，然后串连起来：



代码：

#include <iostream>

#include <vector>

#include <string>

using namespace std;

typedef struct list\_node List;

struct list\_node

{

struct list\_node\* next;

int value;

};

void print\_list(List\* list)

{

List\* tmp=list;

while(tmp != NULL)

{

cout<<tmp->value<<endl;

tmp = tmp->next;

}

}

/\*

初始化List 将从1~n的数字插入到链表中

\*/

void Init\_List(List\*& head,int\* array,int n)

{

head = NULL;

List\* tmp;

List\* record;

for(int i=1;i<=n;i++)

{

tmp = new List;

tmp->next = NULL;

tmp->value = array[i-1];

if(head == NULL)

{

head = tmp;

record = head;

}

else

{

record->next = tmp;

record = tmp;

}

}

}

//求链表长度

int Len\_list(List\* list)

{

if(list == NULL)

return 0;

else

return Len\_list(list->next)+1;

}

/\*

思路：类似于快排，由于要求不改变原来的相对顺序，所以必须有节点的交换

要不然之间交换节点内部的值即可。

\*/

/\*

类似于快速排序的分割

\*/

void PartitionList(List\*& list,int key)

{

if(list == NULL)

return ;

List\* record,\*cur,\*pre,\*tmp;

record = NULL;

cur = list;

pre = NULL;

//开始处理链表

while(cur != NULL)

{

if(cur->value< key) //插入到pre之后，需要特殊处理头节点

{

tmp = cur->next;

if(pre == NULL)

pre = cur;

if(record ==NULL)

{

record = list;

list = cur;

cur->next = record;

record = cur;

pre->next = tmp;

}

else

{

if(pre != record)

{

cur->next = record->next;

record->next = cur;

pre->next = tmp;

record = cur;

}

else

{

record = pre= cur;

}

}

cur = tmp;

}

else

{

pre= cur;

cur = cur->next;

}

}

}

//第二种方法 使用伪指针

List\* PartitionList2(List\* list,int key)

{

List\* head = new List;

head->next = list;

List\* temp = head;

List\* pre = head,\*cur = list,\*next;

//开始处理

while(cur != NULL)

{

next = cur->next;

if(cur->value < key)

{

pre->next = next;

cur->next = temp->next;

temp->next = cur;

temp = cur;

cur = next;

}

else

{

pre = cur;

cur = next;

}

}

return head->next;

}

int main()

{

int array[]={5,1,2,7,8,4,3,6,10,9};

List\* list;

Init\_List(list,array,sizeof(array)/sizeof(int));

list = PartitionList2(list,5);

print\_list(list);

return 0;

}

**题目：**已知链表头指针head与数值x，将所有小于x的节点放在大于或等于x的节点前，且保持这些节点的原来的相对位置。

注：Leetcode 86



**思路：巧用临时头结点**

1. 设置两个节点less\_head和more\_head，分别存储小于和大于的数字：



2、循环：



连接两个临时链表：



**代码：**



**测试：**



### 合并零之间的节点

给你一个链表的头节点 head ，该链表包含由 0 分隔开的一连串整数。链表的 开端 和 末尾 的节点都满足 Node.val == 0 。

对于每两个相邻的 0 ，请你将它们之间的所有节点合并成一个节点，其值是所有已合并节点的值之和。然后将所有 0 移除，修改后的链表不应该含有任何 0 。

返回修改后链表的头节点 head 。

注：

LeetCode 2181

## 多链表分割/合并

### 两数相加

给你两个非空的链表，表示两个非负的整数。它们每位数字都是按照逆序的方式存储的，并且每个节点只能存储一位数字。

请你将两个数相加，并以相同形式返回一个表示和的链表。

你可以假设除了数字 0 之外，这两个数都不会以 0 开头。

注：LeetCode 2

### 两数相加II

给你两个 非空 链表来代表两个非负整数。数字最高位位于链表开始位置。它们的每个节点只存储一位数字。将这两数相加会返回一个新的链表。

你可以假设除了数字 0 之外，这两个数字都不会以零开头。

注：LeetCode 445

### [奇偶链表](https://leetcode-cn.com/problems/odd-even-linked-list)

给定单链表的头节点 head ，将所有索引为奇数的节点和索引为偶数的节点分别组合在一起，然后返回重新排序的列表。

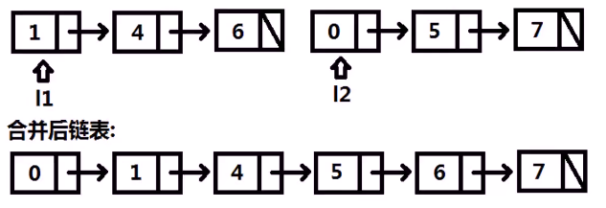
第一个节点的索引被认为是奇数， 第二个节点的索引为偶数，以此类推。

### 单链表的合并/合并两个排序链表

**题目：**

将两个升序链表合并为一个新的 升序 链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

**题目：**有多个已经排序好的单链表，将这些单链表合并成一个有序的单链表，返回合并后的头结点。



注：

Leetcode 21

剑指offer 面试题17

分析：采用构造虚拟节点/哑结点（dummyNode）的方式（可以采用new或者声明），然后构造一个临时节点指向dummyNode，用这个tmpNode去组装新的链表，返回dummyNode->next即可。

**思路：**

比较l1和l2指向的节点，将较小的节点插入到pre指针后，并向前移动较小节点对应的指针。

初始：



循环：



最终结果：



**代码：**



**测试：**



**代码：**

#include <iostream>

#include <vector>

using namespace std;

/\*

合并K个已经排序好的链表

\*/

typedef struct list\_node List;

struct list\_node

{

struct list\_node\* next;

int value;

};

void print\_list(List\* list)

{

List\* tmp=list;

while(tmp != NULL)

{

cout<<tmp->value<<endl;

tmp = tmp->next;

}

}

/\*

初始化List 将从1~n的数字插入到链表中

\*/

void Init\_List(List\*& head,int\* array,int n)

{

head = NULL;

List\* tmp;

List\* record;

for(int i=1;i<=n;i++)

{

tmp = new List;

tmp->next = NULL;

tmp->value = array[i-1];

if(head == NULL)

{

head = tmp;

record = head;

}

else

{

record->next = tmp;

record = tmp;

}

}

}

//求链表长度

int Len\_list(List\* list)

{

if(list == NULL)

return 0;

else

return Len\_list(list->next)+1;

}

/\*

可以类似于合并几个已经排序好的数组这种思想，比如借助最小堆，每次将堆顶元素插入新的链表中，但是也可以使用数组来进行标记，每次从数组中挑选最小的节点直到所有链表的节点都被选中

\*/

List\* Merge\_k(vector<List\*>& vec)

{

int i;

//使用数组进行标记

vector<int> flag(vec.size(),1);

List\* head = NULL;

List\* cur = NULL;

List\* tmp = NULL;

int pos;

//开始处理K个链表

while(1)

{

tmp = NULL;

//开始遍历标记数组，找到合适的节点

for(i =0;i<vec.size();i++)

{

if(vec[i] == NULL)

flag[i] = 0;

if(flag[i])

{

if(tmp ==NULL)

{

tmp = vec[i];

pos =i;

}

if(tmp && tmp->value > vec[i]->value)

{

tmp = vec[i];

pos =i;

}

}

}

//开始特殊处理头节点

if(head == NULL)

{

head = tmp;

cur = head;

}

else

{

cur->next = tmp;

cur= cur->next;

}

//处理被选中的链表

vec[pos] = vec[pos]->next;

if(vec[pos] == NULL)

flag[pos] =0;

//判断是否所有的节点都已经被选中

for(i=0;i<flag.size();i++)

{

if(flag[i])

break;

}

if(i>=flag.size())

break;

}

return head;

}

int main()

{

int array1[]={1,4,7,8,13,19};

int array2[]={5,8,9,10,12,15,17,22,23};

int array3[]={3,6,11,16,17,18,21,24};

int array4[]={2,14,20,25};

vector<List\*> vec(4);

int i;

Init\_List(vec[0],array1,sizeof(array1)/sizeof(int));

Init\_List(vec[1],array2,sizeof(array2)/sizeof(int));

Init\_List(vec[2],array3,sizeof(array3)/sizeof(int));

Init\_List(vec[3],array4,sizeof(array4)/sizeof(int));

List\* head = Merge\_k(vec);

print\_list(head);

return 0;

}

注：2个/K个排序链表的归并

### 排序链表合并（多个）/合并k个升序链表

**题目：**已知k个已排序链表头结点指针，将这k个链表合并，合并后仍为有序的，返回合并后的头结点。



注：Leetcode 23

**分析：**

**方法一：**暴力合并

最普通的方法，k个链表按顺序合并k-1次。

设有k个链表，平均每个链表有n个节点，时间复杂度：

(n+n)+(2n+n)+((k-1)n+n)=(1+2+…+k-1)n=(1+2+…+k)n-n=(k^2+k-1)/2\*n=O(k^2\*n)



**方法二：**排序后相连

将k\*n个节点放到vector中，再将vector排序，再将节点顺序相连。

设有k个链表，平均每个链表有n个节点，时间复杂度：

kN\*logkN+kN=O(kN\*logkN)（比如k=100,n=10000）logkN=20,k=100



**方法三：分治+合并**

对k个链表进行分治，两两进行合并。

设有k个链表，平均每个链表有n个节点，时间复杂度：

第1轮，进行k/2次，每次处理2n个数字；

第2轮，进行k/4，每次处理4n个数字；

最后一次，进行k/(2^logk)次，每次处理2^logk\*N个值。

2N\*k/2+4N\*k/4+8N\*k/8+…+2^logk\*N/(2^logk)=Nk+Nk+…+Nk=O(kNlogk)



**代码：**

**方法一：**



**方法二：**



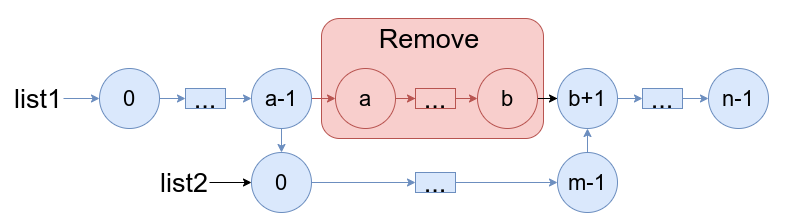
**测试：**

### 合并两个链表

给你两个链表 list1 和 list2 ，它们包含的元素分别为 n 个和 m 个。

请你将 list1 中下标从 a 到 b 的全部节点都删除，并将list2 接在被删除节点的位置。

下图中蓝色边和节点展示了操作后的结果：



请你返回结果链表的头指针。

注：

LeetCode 1669

## 复杂链表深度拷贝

**题目：**已知一个复杂的链表，节点中有一个指向本链表任意某个节点的随机指针（也可以为空），求这个链表的深度拷贝（即构造一个全新的链表而对原来的链表没有影响）。



注：Leetcode 138/剑指offer 面试题26

**分析：**

使用STL map的思想实现映射：



节点地址与节点序号对应（主要是Random指针的赋值）：



**代码：**



**测试：**



## 链表节点删除

**说明：**链表中节点的查找和删除实际上还是链表遍历的拓展。

删除链表的节点，本质上是链表的遍历操作，可以采用**快慢指针**的方式。

**题目要求：**

删除单链表第K个节点

递归打印单链表的节点

判断单链表中是否有环

O(1)时间复杂度删除单链表中的节点

代码：

#include <iostream>

using namespace std;

typedef struct list\_node ListNode;

struct list\_node

{

struct list\_node\* next;

int value;

};

/\*

初始化List 将从1~n的数字插入到链表中

\*/

void Init\_List(ListNode\*& head,int\* array,int n)

{

head = NULL;

ListNode\* tmp;

ListNode\* record;

for(int i=1;i<=n;i++)

{

tmp = new ListNode;

tmp->next = NULL;

tmp->value = array[i-1];

if(head == NULL)

{

head = tmp;

record = head;

}

else

{

record->next = tmp;

record = tmp;

}

}

}

void print\_list(ListNode\* list)

{

ListNode\* tmp = list;

while(tmp != NULL)

{

cout<<tmp->value<<endl;

tmp = tmp->next;

}

}

### 删除链表中的结点

请编写一个函数，使其可以删除某个链表中给定的（非末尾）节点，你将只被给定要求被删除的节点。

注：LeetCode 237

分析：

class Solution {

public:

void deleteNode(ListNode\* node) {

node->val = node->next->val;

node->next = node->next->next;

}

};

### O(1)时间复杂度删除单链表中的节点

/\*

给出一单链表头指针phead和一个待删除的节点指针，

在O(1)时间复杂度内删除此节点

\*/

void Delete(ListNode\* phead,ListNode\* tobedelete)

{

if(tobedelete == NULL || phead == NULL)

return;

ListNode\* temp = phead;

//将下一个节点的数据复制到本节点，然后删除下一个节点

if(tobedelete->next != NULL)

{

tobedelete->value = tobedelete->next->value;

ListNode\* temp = tobedelete->next;

tobedelete->next = tobedelete->next->next;

delete temp;

}

else //要删除的是最后一个节点

{

if(phead == tobedelete)//链表中只有一个节点的情况

{

phead = NULL;

delete tobedelete;

}

else

{

ListNode\* pnode = phead;

while(pnode->next != tobedelete)//找到倒数第二个节点

pnode =pnode->next;

pnode->next = NULL;

delete tobedelete;

}

}

}

int main()

{

int array[]={1,2,3,4,5,6,7,8,9,10};

ListNode\* list;

Init\_List(list,array,sizeof(array)/sizeof(int));

// print\_list(list);

Delete(list,list->next->next->next);

print\_list(list);

return 0;

}



### 删除链表的倒数第 N 个结点

给定一个链表，删除链表的倒数第 n 个节点，并且返回链表的头结点。

注：LeetCode 19

分析：

### 删除链表的中间节点

给你一个链表的头节点head。删除链表的中间节点，并返回修改后的链表的头节点head。

长度为n链表的中间节点是从头数起第⌊n / 2⌋个节点（下标从0开始），其中⌊x⌋表示小于或等于x的最大整数。

对于n = 1、2、3、4和5的情况，中间节点的下标分别是0、1、1、2和2。

注：LeetCode 2095

分析：

使用快慢指针。

### 删除单链表中重复的元素

题目要求：在一个已经排序好的链表中，删除链表中的重复元素，使得重复元素只保留一个。

注：LeetCode 82

分析：

使用快慢指针。

代码：

#include <iostream>

#include <vector>

using namespace std;

typedef struct list\_node List;

struct list\_node

{

int value;

struct list\_node\* next;

};

void Init\_List(List\*& head,int\* array,int n)

{

head = NULL;

List\* tmp;

List\* record;

for(int i=1;i<=n;i++)

{

tmp = new List;

tmp->next = NULL;

tmp->value = array[i-1];

if(head == NULL)

{

head = tmp;

record = head;

}

else

{

record->next = tmp;

record = tmp;

}

}

}

void print\_list(List\* list)

{

List\* tmp=list;

while(tmp != NULL)

{

cout<<tmp->value<<endl;

tmp = tmp->next;

}

}

/\*

删除重复的元素

\*/

void RemoveDuplicate(List\*& head)

{

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

return;

List\* slow = head;

List\* fast = head->next;

List\* tmp;

while(fast != NULL)

{

if(slow->value == fast->value)

{

tmp = fast;

fast = fast->next;

slow->next = fast;

delete tmp;

}

else

{

slow = slow->next;

fast = fast->next;

}

}

}

//另一种方法

List\* removeDuplicate(List\* head)

{

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

return head;

List\* temp,\*pre,\*cur;

List\* newhead = new List;

newhead->next = head;

temp = newhead;

pre = head;

cur = head->next;

while(cur != NULL)

{

while(cur != NULL && cur->value == pre->value)

cur = cur->next;

if(pre->next == cur)

{

temp->next = pre;

temp = pre;

pre = cur;

}

else

{

pre = cur;

}

if(cur != NULL)

cur = cur->next;

}

temp->next = cur;

return newhead->next;

}

int main()

{

int array[]={1,1,1,2,3,3,4,5,6,6,7,7};

List\* head;

Init\_List(head,array,sizeof(array)/sizeof(int));

head = removeDuplicate(head);

print\_list(head);

return 0;

}

**拓展：**

在一个已经排序号的链表中，删除链表中的重复元素，使得重复元素不保留。

### 移除链表元素/重复元素

说明：LeetCode 203（类似删除重复元素）

### 从链表中移除元素

给你一个链表的头节点 head 。

移除每个右侧有一个更大数值的节点。

返回修改后链表的头节点 head 。

注：LeetCode 2487

### 从链表中删去总和值为零的连续节点

给你一个链表的头节点 head，请你编写代码，反复删去链表中由 总和 值为 0 的连续节点组成的序列，直到不存在这样的序列为止。

删除完毕后，请你返回最终结果链表的头节点。

注：LeetCode 1171

分析：

class Solution {

public:

ListNode\* removeZeroSumSublists(ListNode\* head) {

ListNode\* dummy = new ListNode(0);

dummy->next = head;

int prefix = 0;

unordered\_map<int, ListNode\*> seen;

for (ListNode\* node = dummy; node; node = node->next) {

prefix += node->val;

seen[prefix] = node;

}

prefix = 0;

for (ListNode\* node = dummy; node; node = node->next) {

prefix += node->val;

node->next = seen[prefix]->next;

}

return dummy->next;

}

};

## 双向链表

### LRU缓存

注：LeetCode 146

### LFU缓存

## 通讯录

## 一元多项式

## 约瑟夫环

题目要求：使用单链表来解决约瑟夫环问题。

代码：

#include <iostream>

#include <vector>

#include <list>

using namespace std;

// 约瑟夫环问题，使用STL中的List，

int Joseph(list<int>& ring,int k)

{

list<int>::iterator itr = ring.begin(),temp;

int m;

//链表节点个数大于1就会一直删除下去

while(ring.size()>1)

{

m =1;

//判断当前迭代器是否需要重新置位

if(itr == ring.end())

itr = ring.begin();

//查找合适的节点进行删除

while(m <k)

{

//始终需要判断是否需要重新置位

if(itr == ring.end())

itr = ring.begin();

itr++;

m++;

if(itr == ring.end())

itr = ring.begin();

}

temp = itr;

itr++;

//删除被选中的节点

ring.erase(temp);

}

//只剩下一个节点

itr = ring.begin();

return \*itr;

}

int main()

{

int i;

list<int> ring;

for(i=1;i<=9;i++)

ring.insert(ring.begin(),i);

cout<<Joseph(ring,5)<<endl;

return 0;

}