# 2023第一次研讨课

学号：2256666 姓名：刘禹锡

1. **问题一**
2. **题目描述**

给定两个单向链表的头指针，判定它们是否相交(为简化问题，假设两个链表均不带环)

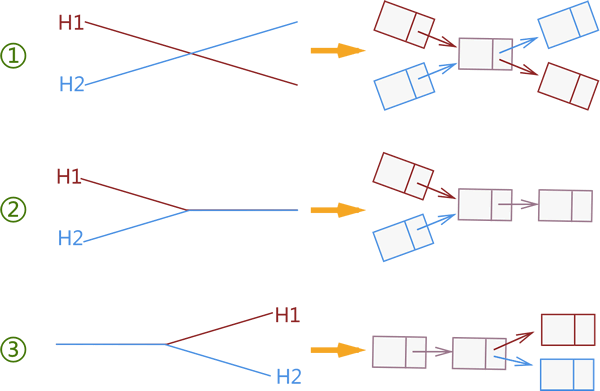
1) 请画出链表相交示意图

2) 给出算法思想，并分析时间复杂度

3) 如何求出两个单链表的第一个交点

1. **理解与分析**

这道题考察的是单向链表的知识点，单向链表由于没有索引，仅靠指针链接，所以只可以顺序访问。由于本题单链表不存在环，所以初步推断相交链表可能有以下形式



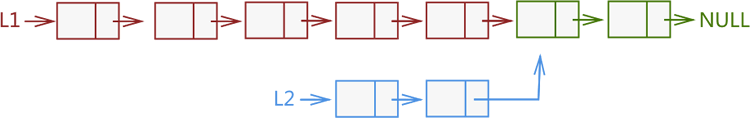
但是通过画图分析可以发现，单链表仅有一个指针，只可以指向一个节点，如果相交，则后续节点也必定重合，所以仅存在②一种情况。而对于这道题，已知两个单向链表（不含环）的头指针，如何判断是否相交，考虑到三种比较有代表性的算法。

1. **暴力遍历**

大体思想就是由头指针循环遍历两个链表，判断是否会出现指针指向相同的节点。如果出现，那么记录下第一个，就是相交的第一个节点。但是这样最坏情况需要嵌套循环遍历两个链表，时间复杂度较高，为O(n2)。

1. **遍历优化**

通过图示我们可以发现，如果两个链表相交，则后续节点一定相同，并且后续长度和尾节点都是是相同的，所以我们可以先向右对齐链表，同时比较最后的节点，如果相同则说明相交，记录下长度，再将长链表的指针移动差值个单位，实现对齐，再循环遍历，寻找第一个相交节点。时间复杂度为O(n).与上述暴力遍历的区别在于，我们在长链中找到了与短链表等长的子链，再循环遍历时，两者的指针可以同时移动，而不是定一动一。



1. **运用栈**

要判断尾结点是否相同，我们可以利用栈的“后进先出”特性来简化操作。为此，我们可以创建两个栈，将两个链表中的所有节点地址依次入栈。然后，弹出栈顶元素，并比较它们的地址，以判断它们是否相交。而后，寻找第一个相交节点，只需要依次出栈比较即可，时间复杂度也为O(n)。

1. **结构与算法**

单链表结构实现：

typedef struct Link {

char elem; //数据域

struct Link \* next; //指针域，指向直接后继元素

}link; //link为节点名，每个节点都是一个 link 结构体

1. **暴力遍历**

Link\* linkOrNo (link \* L1, link \* L2) {

link \* p1 = L1;

link \* p2 = L2;

while (p1) //逐个遍历 L1 链表中的各个节点

{

//遍历 L2 链表，针对每个 p1，依次和 p2 所指节点做比较

while (p2) {

if (p1 == p2) {

return p1;

}

p2 = p2->next;

}

p1 = p1->next;

}

return NULL;

}

本函数传入两个单链表的头指针，然后使用嵌套循环，一对一比较节点是否相同，如果相同返回第一个相同的节点，如果不同，返回NULL。

由于存在嵌套循环，时间复杂度为O(n2)

1. **遍历优化**

**获取链表长度和尾结点：**

/\* description: 计算链表的长度并返回尾结点

Param head 用指针形式传入链表的头结点

Param tail 使用引用形式传出尾结点

return len 返回链表长度\*/

int getLengthAndTail(link\* head, link\* &tail) {

int len = 0;

link\* curr = head;

while (curr) {

len++;

if (!curr->next) {

tail = curr; // 记录尾结点

}

curr = curr->next;

}

return len;

}

本函数使用while循环，通过遍历的方法记录链表长度，以及尾结点。

存在一个循环，时间复杂度为O(n)

**判断是否相交并获取首个相交节点：**

/\* description: 通过比较尾结点判断是否相交并获取首个相交节点

Param L1 L2 用指针形式传入链表的头结点

return 返回首个相交节点

\*/

link\* linkOrNo (link\* L1, link\* L2) {

---初始化尾结点，并调用上述函数获取长度与尾结点---

if (尾结点不同) { 返回空 }

---不满足if，说明相交，因此初始化头结点，用于下面遍历寻找首个相交节点---

int lengthDiff = abs(len1 - len2); // 计算长度差

if (len1 > len2) { // 向右对齐

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

p1 = p1->next;

} else {

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

p2 = p2->next;

}

// 遍历链表，找到第一个相交节点

while (p1 && p2) {

if (p1 == p2) {

return p1; // 返回相交节点

}

p1 = p1->next;

p2 = p2->next;

}

return NULL; // 如果没有相交节点，返回NULL

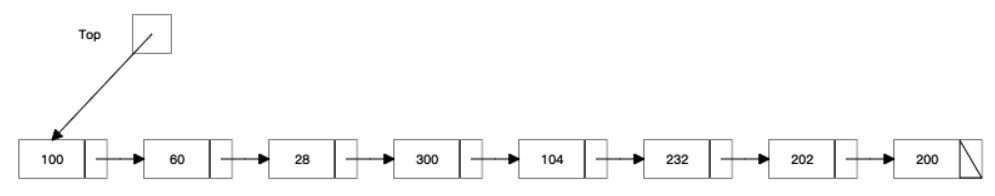
}

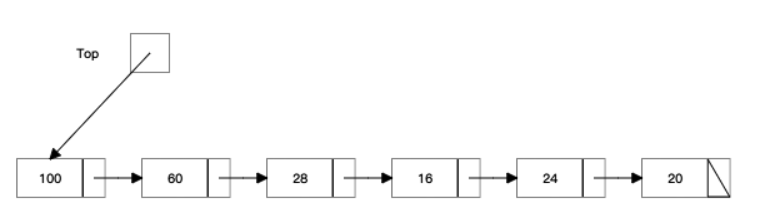
本函数首先判断是否相交，若相交，则首先根据长度差按尾部对齐，然后创建两个指针，分别指向首个对齐节点，然后同步移动，一一比较，只到找到第一个相同位置，即为首个相交节点。

主要时间开销为对齐后遍历链表的循环，时间复杂度为O(min（len1，len2）)，简化为O(n).

1. **运用栈**

**本方法使用栈的思想，首先需要构建栈的数据结构：**

**Link1**

**Link2**

由上图我们可以看出，通过遍历两个链表将其节点入栈，我们可以实现尾端对齐，这实际上与上一个算法类似，然后我们只需要比较栈顶节点是否相同即可，显然，栈顶元素都为100，所以两个链表相交。

接下来，我们可以依次出栈，过程如下：link1 pop 60,link2 pop 60,60=60，first=60；link1 pop 28，link2 pop 28，28=28，first=28；link1 pop 300，link2 pop 16，16 != 300，所以首个相交节点最终为28.

通过上面的过程，我们可以发现时间消耗主要在读入栈（O(n)）以及判定相交后的出栈（O(n)），所以总体的时间复杂度为O(n).

**算法实现：**

node temp=NULL; //存第一个相交节点

while(!stack1.empty()&&!stack1.empty()) //两栈不为空

{

temp=stack1.top(); //存栈顶，即临时的相交节点

stack1.pop(); //同时出栈并比较

stack2.pop();

if(stack1.top()!=stack2.top()){

break;

}

}

1. **效率讨论**

通过上面的讨论分析，可以清楚地看到，对于暴力破解算法，时间复杂度最高，为O(N2),如果想要降低时间复杂度，就需要观察分析单链表其特点，可以发现，如果相交，必定后续节点重合，因此，可以直接判断尾结点，这样就把n\*n变成了n+n，时间复杂度降低了一个层次。在算法的具体实现上，较为灵活，我们可以直接通过循环，也可以使用有相似特点的栈来实现。

同时，本题还要考虑找到第一个相交节点，就不能简单地遍历到末尾，还需要存储一些变量来帮助接下来的查找。

优化：经过网络搜索，发现还可以使用哈希表存储出现的地址，然后在比较两者字典是否相同。

1. **问题二**
2. **题目描述**

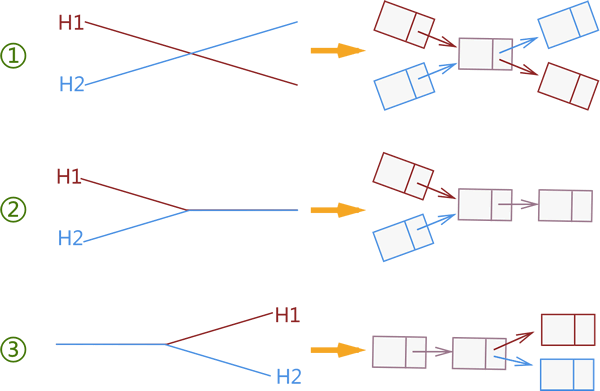
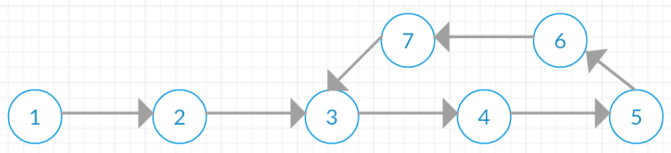
如何判断一个单链表内是否有环?若有环，找到入环的第一个节点。

1. **理解与分析**

本题让我们判断单链表是否带环，并且找到环的入口点。对于这个问题，首先分析带环单链表的结构，可以发现单链表仅可以带一个环。带环->带圆->首尾相接->可以循环。由此生发出几个较为典型的思路。

1. **判断相交**

受上题的启发，我们可以发现，如果两个单链表相交，那么会出现三个端点，两个头结点和一个尾结点，如果将任意一个头结点与尾结点连接，那么就形成了一个带环单链表。可以使用这个思路判断相交来判断成环，或者在判定成环后劈开，调用上面的算法获取入环节点。

1. **穷举遍历**

因为带环的单链表可以循环回来，在遍历的过程中总会遇到之前访问过的节点。

所以采取穷举遍历法，思路：首先从头节点开始，依次遍历单链表的每一个节点。每遍历到一个新节点，就从头节点重新遍历新节点之前的所有节点，用新节点ID和此节点之前所有节点ID依次作比较。如果发现新节点之前的所有节点当中存在相同节点ID，则说明该节点被遍历过两次，链表有环；如果之前的所有节点当中不存在相同的节点，就继续遍历下一个新节点，继续重复刚才的操作。

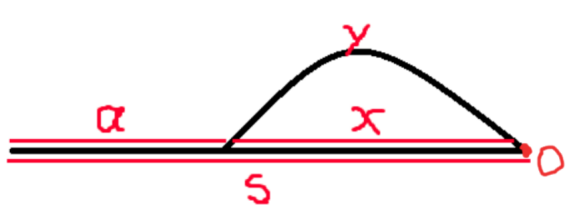
但是穷举遍历，需要每次都从头结点循环，效率有些低。

1. **快慢指针法**

受跑圈问题启发，单链表带环，形似跑道，我们可以定义两个指针，一个fast，每次走两步，一个slow，每次走一步，经过一段时间，两个指针都会进入环中并且不会出来，由于fast比slow快，所以经过若干回合，fast总会追上slow，就说明单链表带环。

下面进行数学验算。相遇点记为O，慢指针已走的环路程记为x，环剩下的路程记为y。

设slow在相遇前走了s步，则fast走了2s步，设环长为r，有2s=s+nr，即s=nr。由上图可知a+x=s, x+y=r，而我们的目标是找到a的位置。a+x=s=nr=(n-1)r+r=(n-1)r+y+x，则a=(n-1)r+y，从链表头和相遇点O分别设一个指针，每次各走一步，这两个指针必定相遇，且相遇的第一个点为环入口点。



1. **Set集合大小变化**

用 set 遍历链表，把节点放入set里，每次访问下个节点时，如果set长度不变，则跳出，说明有环。否则set长度+1，继续遍历。该方法时间复杂度是O（N），空间复杂度上因为需要额外等数量的存储空间，所以空间复杂度是O（n）.

1. **逻辑结构与代码**

// 创建一个有环的链表作为示例

LinkNode \* head = new LinkNode(1);

LinkNode \* second = new LinkNode(2);

LinkNode \* third = new LinkNode(3);

head->next = second;

second->next = third;

third->next = second; // 创建环

1. **穷举遍历法**

bool hasCycle(LinkNode \* head) {

LinkNode \* current = head;

while (current) {

LinkNode \* runner = head;

// 遍历当前节点之前的所有节点，查看是否有相同的节点

while (未发现相同节点) {

if (runner == current->next)

return true; // 发现相同节点，链表有环

runner = runner->next;

}

current = current->next;

}

return false;

}

原理同前，由于嵌套循环，算法的时间复杂度是0+1+2+3+…+(D+S-1) = (D+S-1)\*(D+S)/2

时间复杂度为O(n2).

1. **快慢指针法**

**判断是否有环：**

/\*\*

\* 判断单链表是否存在环

\* @param head

\* @return

\*/

boolean isLoopList(link\* head){

link\* slow, fast;

//使用快慢指针，慢指针每次向前一步，快指针每次两步

slow = fast = head;

while(fast不空并且fast.next不空){

slow = slow.next;

fast = fast.next.next;

//两指针相遇则有环

if(slow == fast)

return true;

}

return false;

}

原理同前，时间复杂度为O(n)

**找到入口点：**

/\*\*

\* 找到有环链表的入口

\* @param head

\* @return

\*/

Link \* findEntrance (Link \* head){

---找环操作，同上---

//一个指针从链表头开始，一个从相遇点开始，每次一步，再次相遇的点即是入口节点

if(存在环){

slow = head;

while(fast不空并且fast.next不空){

//两指针相遇的点即是入口节点

if(slow == fast){

return slow;

}

slow = slow.next;

fast = fast.next;

}

}

return null;

}

原理同前，时间复杂度为O(n).相较于穷举快速了不少。

1. **Set集合大小变化**

set 是C++标准库中的一个关联容器，它是一个有序集合，用于存储一组唯一的元素。我们也可以通过自定义结构来实现。

bool hasCycle(LinkNode\* head) {

std::set<LinkNode\*> visitedNodes;//使用set集合储存已访问数据

LinkNode\* current = head;

while (current) {

if (visitedNodes.find(current) != visitedNodes.end()) {

// 如果当前节点已经在集合中，说明存在环

return true;

}

visitedNodes.insert(current); //不存在则插入集合

current = current->next;

}

// 如果链表遍历结束都没有发现环，则链表无环

return false;

}

原理同前，主要使用了set有序集合的特点，时间复杂度为O(n).

1. **效率讨论**

通过上述几种方法，实现了判断单链表是否有环并且找到环的入口的功能。比较这几种方法，可以发现穷举遍历的时间复杂度为O(N2),为时间开销最大的，这是因为它采取一种两个两个相互比较的形式。而其他方法，最具技巧性的就是快慢指针法，它利用了环的循环结构，通过快慢指针相遇判断是否存在环以及环入口的定位操作。另一种则采用了set集合，通过比较元素是否已包含来判断是否含环，类似于哈希表，这两个的时间复杂度都为O(N),因为他们的链表遍历都是同步的。

1. **问题三**
2. **题目描述**

开放性题目：医院看病排队管理，不考虑急诊。

1. **理解与分析**

本题和银行排队有些类似，并且不用考虑急诊，不存在优先级问题。如果不考虑特殊情况，这个系统应该满足的基本操作有入队，出队，查询操作。所以打算使用队列来实现。

另外，医院有不同科室，同一科室又有不同医生，所以本题采用一般性描述，存在三个科室，每个科室又有两个医生，假设患者选择随机，分配医生时，优先分配到短队列。

所以，本系统需要实现功能如下：患者的创建与入队，包含患者所选择的科室，按照患者选择的科室再分配医生，将其入队；患者就诊：通过叫号命令将患者出队就诊；患者查询：根据患者的编号查询所在科室与前面人数。

1. **逻辑结构与算法**

对于这种问题，在实际情况下变动较多，很有可能由于技术政策变化（疫情，预约排队）需要修改，所以使用面向对象的思想将很有助于系统的迭代升级，因此，创建患者结构体，存放患者编号，以及要看的科室信息；创建医院队列类，用于存放队列信息。同时，为了方便，使用STL库中的vector和queue来模拟。

以下为算法实现:

患者结构体：

struct Patient {

string name;

int department; // 科室编号，0表示第一科室，1表示第二科室，2表示第三科室

int doctor; // 医生编号，0表示第一个医生，1表示第二个医生

Patient\* next;

Patient(const string& n, int d, int doc) : name(n), department(d), doctor(doc), next(NULL) {}

};

队列实现：

class LinkedListQueue {

private:

Patient\* front;

Patient\* rear;

public:

LinkedListQueue() : front(NULL), rear(NULL) {} //初始化队首队尾

void enqueue(const string& name, int department, int doctor) { //入队

Patient\* patient = new Patient(name, department, doctor); //创建节点

if 判空处理

else { 加到队尾 }

bool dequeue(string& name, int& department, int& doctor) { //出队

---判空---

-----------

front = front->next;

---判空---

delete patient;

}

bool isEmpty() const { 判空 }

~LinkedListQueue() { 释放队列 }

};

入队出队判空函数的时间复杂度均为O(1)，因为只执行常数时间的操作。

医院排队实现：

class HospitalQueue {

private:

LinkedListQueue patientQueues[3][2]; //三个科室，每个科室两个医生

int selectDoctor(int department) { 选择医生，按照医生队列长度，优先选取人数少的 }

public:

HospitalQueue() { 动态内存申请，为每一个医生创建一个队列 }

void enqueuePatient(const string& name, int department) { //患者入队

int selectedDoctor = selectDoctor(department); //获取选择医生与诊室

patientQueues[department][selectedDoctor].enqueue(name, department, selectedDoctor);

输出相关提示

}

void treatNext () { //接诊下一个病人

for (int department = 0; department < 3; ++department) {

for (int doctor = 0; doctor < 2; ++doctor) {

string name;

int dep, doc;

if (出队成功) { 输出相关提示 }

}

}

}

void queryPatient(int department, const string& name) { //查找病人，并获得前面等待人数

按照科室以及病人姓名查找，返回就诊医生和前面等待人数。

采用方法为遍历队列的方法。

}

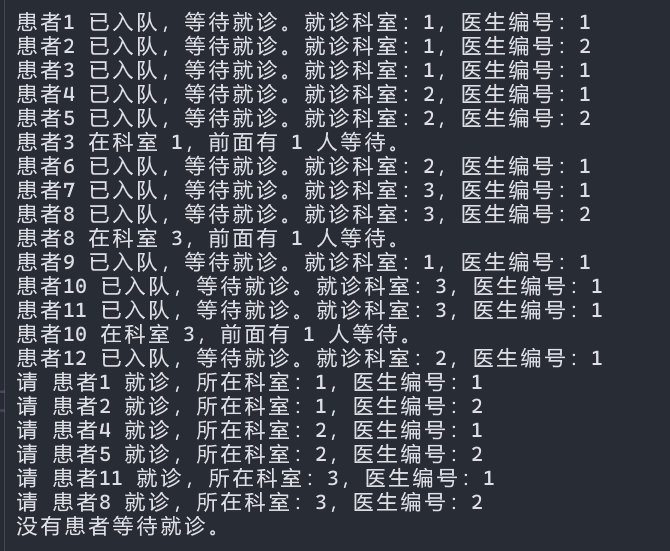
};

enqueue将患者入队操作依赖LinkedListQueue决定，时间复杂度为O(1)

treatNext接诊各个诊室的患者操作实际为将三个科室6个医生的队首元素出队，O(3\*2)为常数时间，所以时间复杂度为O(1).

queryPatient方法根据科室遍历两个医生，最坏情况需要遍历两个队列，时间复杂度为O(n).

运行效果如下图：

****

1. **效率讨论**

这道题考察队列的实际应用，尽管考虑了许多因素，但是还是理想化了许多现实因素，例如终止离开，看病时长等。此外，由于医院人流量大，对于算法的复杂度要求要高很多，而上述算法利用队列，使得病人“FIFO”，时间复杂度也都在O(n)及以下，效率较高。

另外，之所以采用链表队列是由于链表利于动态内存申请，并且方便增删改操作。此外，我尝试使用了面向对象的方法，便于后续优化等。