**第一次研讨课报告**

1. **判断两个链表是否相交，若相交，求出第一个相交的节点**

相交定义：两个链表相交意味着它们在某个节点开始共享后续的所有节点。

以下是几个方法：

**解决方案 1：长度差法**

通过计算两个链表的长度，调整较长链表的起始位置，使得两个链表从同一位置开始比较：

1. 计算两个链表的长度。
2. 计算长度差，并将较长的链表向前移动差值步数。
3. 同时遍历两个链表，找到第一个相交的节点。

int getLength(ListNode \* head) {

int length = 0;

while (head) {

length++;

head = head->next;

}

return length;

}

ListNode\* findIntersection(ListNode \* headA, ListNode \* headB) {

int lenA = getLength(headA);

int lenB = getLength(headB);

// 调整起始位置

while (lenA > lenB) {

headA = headA->next;

lenA--;

}

while (lenB > lenA) {

headB = headB->next;

lenB--;

}

// 同时遍历两个链表

while (headA && headB) {

if (headA == headB) {

return headA; // 返回第一个相交节点

}

headA = headA->next;

headB = headB->next;

}

return nullptr; // 无相交

}

**时间复杂度**：O(m + n)，其中m和n分别为两个链表的长度。

**解决方案 2：双指针法**

使用双指针遍历两个链表，分别指向每个链表的起始位置。当一个指针到达链表末尾时，转向另一个链表的起始位置。两个指针会在相交节点相遇：

1. 初始化两个指针指向两个链表的头节点。
2. 遍历两个链表，若到达末尾则切换到另一个链表。
3. 如果两个指针相遇，则返回相交节点；如果都为nullptr，则返回nullptr。

ListNode\* findIntersection(ListNode \* headA, ListNode \* headB) {

if (!headA || !headB) return nullptr;

ListNode\* ptrA = headA;

ListNode\* ptrB = headB;

// 遍历两个链表

while (ptrA != ptrB) {

// 指向下一个节点，若到达末尾则转向另一个链表

ptrA = (ptrA == nullptr) ? headB : ptrA->next;

ptrB = (ptrB == nullptr) ? headA : ptrB->next;

}

return ptrA; // 返回相交节点，若无相交则为 nullptr

}

**时间复杂度**：O(m + n)，遍历两个链表。

**解决方案 3：转化为循环链表**

1. 遍历第一个链表，找到其尾节点。
2. 将尾节点的next指针指向第二个链表的头节点。
3. 使用下面的方法检测新链表是否有环。

具体实现不列出了

### 总结

以上三种方案均能有效判断两个链表是否相交并找出第一个相交节点：

* **方案 1**使用长度差法，调整起始位置，空间复杂度较低。
* **方案 2**的双指针法是最优选择，能在O(1)空间复杂度下找到相交节点。

根据具体情况选择合适的方案来解决链表相交的问题。

1. **判断一个链表是否有环存在，如有，求第一个入环元素**

在链表中判断是否存在环并找到第一个入环节点。以下是几种解决方案及其算法复杂性分析。

**解决方案 1：布尔标记法**

通过在链表节点中添加一个布尔标记，来跟踪节点是否被访问过：

1. 初始化时，使用新的指针ptr遍历链表。
2. 如果ptr的标记为false，将其改为true并继续遍历。
3. 若发现某节点的标记为true，表示链表有环，并可以追踪到第一个入环元素。

struct ListNode {

int data;

bool visited; // 标记是否被访问过

ListNode\* next;

};

ListNode\* findCycle(ListNode \* head) {

ListNode\* ptr = head; // 新指针，保证 head 不变

while (ptr) {

if (ptr->visited) return ptr; // 返回第一个入环节点

ptr->visited = true;

ptr = ptr->next;

}

return nullptr; // 没有环

}

**时间复杂度：**O(n)，遍历链表一次。

**解决方案 2：顺序表法**

遍历链表时，将访问过的节点地址存储在顺序表中（数组等）。在遍历过程中，每次检查当前节点的地址是否已存在于顺序表中：

1. 遍历链表，将每个节点的地址存入顺序表。
2. 使用线性搜索来检查当前节点是否已在顺序表中。
3. 若存在，则说明链表有环；若不存在，则继续遍历。

#define MAX\_NODES 1000 // 假设链表节点数不超过1000

ListNode\* findCycleStart(ListNode \* head) {

ListNode\* visited[MAX\_NODES] = { nullptr }; // 初始化数组

int index = 0;

ListNode\* ptr = head;

while (ptr) {

// 检查当前节点是否已在数组中

for (int i = 0; i < index; i++) {

if (visited[i] == ptr) {

return ptr; // 返回第一个入环元素

}

}

// 存储当前节点地址

visited[index++] = ptr;

ptr = ptr->next;

}

return nullptr; // 无环

}

**时间复杂度**：O(n^2)，因为每次查找都需线性时间。

#### 解决方案 3：快慢指针法

通过快慢指针的方法找到环，并进一步确定入环元素：

1. 使用快指针和慢指针检测环。
2. 一旦相遇，重置一个指针为头节点，并与相遇的指针一起移动，直到再次相遇。
3. 相遇的节点即为第一个入环元素。

ListNode\* findCycleStart(ListNode \* head) {

ListNode\* slow = head;

ListNode\* fast = head;

// 检测环

while (fast && fast->next) {

slow = slow->next;

fast = fast->next->next;

if (slow == fast) {

// 找到环的起点

slow = head; // 重置慢指针

while (slow != fast) {

slow = slow->next;

fast = fast->next;

}

return slow; // 返回第一个入环元素

}

}

return nullptr; // 无环

}

**时间复杂度**：O(n)，遍历链表

### 总结

以上三种方案有效地判断链表中是否有环，并找出第一个入环元素：

* **方案 2**虽然易于理解，但存在复杂度较高的问题。
* **方案 3**的快慢指针法是最优选择，因为它在不增加额外空间的情况下，能够快速找到入环元素。

根据具体情况选择合适的方案进行链表环的检测和入环元素的查找。

1. **医院看病排队管理**

#### 解决方案1：使用队列的数据结构

为了管理不同病情程度的患者，我们可以为每种优先级创建一个队列。患者结构体包含优先级和姓名，并按照优先级进入队列，出队时则根据优先级进行处理。

**患者结构体**

struct Patient {

int pri; // 病情优先级

string name; // 患者姓名

Patient\* next; // 指向下一个患者（链表实现）

};

队列容器

struct WaitQueue {

WaitQueue() : head(NULL), rear(NULL) {}

Patient\* head; // 链表头

Patient\* rear; // 链表尾

};

vector<WaitQueue> sys(5); // 假设有5个优先级

患者输入与出队

int n; // 病人数量

for (int i = 0; i < n; i++) {

Patient\* pa = new Patient(); // 创建患者对象

// 输入患者信息

cin >> pa->pri >> pa->name;

insert(sys[pa->pri - 1], pa); // 按优先级入队

}

// 按优先级出队

for (int i = 0; i < 5; i++) {

if (sys[i].head) {

cout << sys[i].head->name << endl; // 输出当前优先级的第一个患者

Patient\* temp = sys[i].head;

sys[i].head = sys[i].head->next; // 出队

delete temp; // 释放内存

}

}

链表插入

void insert(WaitQueue & wq, Patient \* pa) {

pa->next = NULL; // 新患者的next指针指向NULL

if (wq.rear) {

wq.rear->next = pa; // 将新患者链接到队列尾部

}

else {

wq.head = pa; // 若队列为空，将新患者设为头

}

wq.rear = pa; // 更新尾部指针

}

**患者信息录入**

for (int i = 0; i < n; i++) {

Patient\* pa = new Patient(); // 创建新的患者

cin >> pa->pri >> pa->name; // 输入患者信息

insert(sys[pa->pri - 1], pa); // 按优先级插入链表

}

#### 解决方案2：统一队列与优先队列

将所有患者统一放入一个优先队列中，根据病情严重程度进行实时排序。在排序过程中，可以使用多关键字排序，以确保在优先级相同的情况下根据到达时间进行排序。

患者结构体及优先级队列

struct Patient {

int priority; // 病情优先级

int time; // 到达时间

Patient(int pri, int t) : priority(pri), time(t) {}

bool operator<(const Patient& p) const {

if (this->priority == p.priority)

return this->time > p.time; // 优先级相同，早到的优先

return this->priority < p.priority; // 优先级高的优先

}

};

priority\_queue<Patient> q; // 使用优先队列

患者信息录入

for (int i = 0; i < n; i++) {

int pri, t;

cin >> pri >> t; // 输入优先级和到达时间

q.push(Patient(pri, t)); // 入队

}

// 医生空闲时出队

while (!q.empty()) {

cout << q.top().name << endl; // 输出当前优先患者

q.pop(); // 出队

}

### 总结

通过以上方案，我们可以有效管理医院中患者的排队情况：

* **方案 1**提供了基于优先级的队列管理，适合简单场景。
* **方案 2**利用优先队列，能在复杂情况下实时排序，提升处理效率。

根据实际需求选择合适的方案进行患者排队管理。