习题解答

1、链表

一、有13个人围成一圈,从第1个人开始顺序报号1,2,3。凡报到3者退出圈子。找出最后留在圈子中的人原来的序号。要求用链表实现

思路:

> 首先定义了一个`Node`结构体来表示链表的节点，每个节点包含一个整数数据和一个指向下一个节点的指针。`createCircularList`函数创建了一个包含n个节点的循环链表，并返回链表的头节点。`findJosephusWinner`函数实现了约瑟夫环的逻辑，即报数到m的人会被删除，直到链表中只剩下一个节点。最后，`main`函数调用这些函数来找出并打印最后留在圈子中的人的序号。

代码:

```c
#include <stdio.h>
#include <stdlib.h>

// 定义链表节点结构体
typedef struct Node {
    int data;           // 存储人的序号
    struct Node* next;  // 指向下一个节点的指针
} Node;

// 创建一个含有n个节点的循环链表
Node* createCircularList(int n) {
    Node *head = NULL, *tail = NULL, *newNode = NULL;
    for (int i = 1; i <= n; i++) {
        newNode = (Node*)malloc(sizeof(Node));
        newNode->data = i;
        newNode->next = NULL;
        if (!head) {
            head = newNode;  // 初始化头节点
        } else {
            tail->next = newNode;  // 将新节点添加到链表末尾
        }
        tail = newNode;
    }
    tail->next = head;  // 使链表成环
    return head;
}

// 找出最后留在圈子中的人的序号
int findJosephusWinner(Node* head, int m) {
    Node *current = head, *previous = NULL;
    while (current->next != current) {  // 当链表中不止一个节点时继续循环
        for (int count = 1; count < m; count++) {  // 报数到m-1
            previous = current;
            current = current->next;
        }
        // 报数到m，删除当前节点
        previous->next = current->next;
```

```
        free(current);  // 释放被删除节点的内存
        current = previous->next;  // 继续从下一个节点开始报数
    }
    int winner = current->data;  // 最后剩下的节点的序号
    free(current);  // 释放最后一个节点的内存
    return winner;
}

int main() {
    int total_people = 13;  // 总人数
    int m = 3;              // 报数到3的人退出圈子
    Node* head = createCircularList(total_people);  // 创建循环链表
    int winner = findJosephusWinner(head, m);  // 找出赢家
    printf("最后留在圈子中的人原来的序号是: %d\n", winner);
    return 0;
}
```

二、 已有a,b两个链表,每个链表中的结点包括学号,成绩. 要求把两个链表合并, 按学号升序排列

```
#include <stdio.h>
#include <stdlib.h>

// 定义链表节点结构体
typedef struct Node {
    int student_id; // 学号
    int score; // 成绩
    struct Node *next; // 指向下一个节点的指针
} Node,* List;



// 合并两个已排序的链表，并在合并前对每个链表进行排序
List mergeSortedLists(List head1, List head2) {
    // 对第一个链表进行插入排序
    InsertSort(head1);
    // 对第二个链表进行插入排序
    InsertSort(head2);

    // 创建一个哑节点，作为合并后链表的起始点
    Node dummy;
    // tail指针初始化为空节点
    List tail = &dummy;
    // 空节点的下一个节点设为NULL
    dummy.next = NULL;

    // 当两个链表都不为空时，合并它们
    while (head1 != NULL && head2 != NULL) {
        // 比较两个链表当前节点的学号，选择较小的节点
        if (head1->student_id < head2->student_id) {
            tail->next = head1; // 将head1连接到tail之后
            head1 = head1->next; // head1移动到下一个节点
        } else {
            tail->next = head2; // 将head2连接到tail之后
```

```
            head2 = head2->next; // head2移动到下一个节点
        }
        tail = tail->next; // 更新tail为合并后链表的最后一个节点
    }

    // 将剩余的节点连接到合并后的链表末尾
    tail->next = (head1 == NULL) ? head2 : head1;

    // 打印合并后的链表
    printList(dummy.next);

    // 返回合并后的链表的头指针
    return dummy.next;
}

// 对链表进行插入排序
void InsertSort(List head) {
    if (head == NULL || head->next == NULL) return; // 如果链表为空或只有一个节点，无需排序

    List sorted = NULL; // 初始化已排序部分的头指针为NULL
    List current = head; // 当前要排序的节点
    List next = NULL; // 保存下一个要排序的节点
    List prev = NULL; // 用于在已排序链表中找到正确的插入位置的前一个节点

    while (current != NULL) {
        next = current->next; // 保存下一个节点

        // 在已排序部分中找到正确的插入位置
        prev = NULL;
        List pos = sorted;
        while (pos != NULL && pos->student_id < current->student_id) {
            prev = pos; // 移动prev到pos
            pos = pos->next; // 移动pos到下一个节点
        }

        // 将current节点插入到已排序部分
        if (prev == NULL) {
            // 插入到sorted的开始
            current->next = sorted;
            sorted = current;
        } else {
            // 插入到prev之后
            current->next = prev->next;
            prev->next = current;
        }

        current = next; // 移动到下一个要排序的节点
    }

    // 更新头指针指向已排序链表的开始
    head = sorted;
}
```