Data Structure Lab Report

实验序号	学号	姓名
Lab1	2021302488	郑安玮

1 问题描述

1. **归并数组**
   给定两个按照升序排列的有序数组，请把它们合成一个升序数组并输出。
2. **高精度计算π值** 限制使用双向链表作存储结构，请根据用户输入的一个整数（该整数表示精确到小数点后的位数，可能要求精确到小数点后 500 位），高精度计算PI值。可以利用反三角函数幂级展开式来进行计算。

2 问题的数学分析

2.1 归并数组

问题要求输入两个升序数组，将两个数组合并后保持单调性，按行输出。

由于问题涉及频繁插入，并且排序算法不需要随机访问，因此采用单链表作为存储结构。

> 基本的算法思路如下：
>
> - 定义三个链表分别用于存储两个输入矩阵和一个输出矩阵。
> - 依次访问两个输入链表的节点，对节点中的数据域进行比较，将较小值加入输出链表，将较小值指针后移。
> - 重复上述步骤，直到某一链表指针为空。
> - 将非空指针后续节点加入输出链表。

2.2 高精度计算π值

问题要求使用密集展开式计算高精度π值，首先看一下反三角函数幂级数展开公式。

$\frac{\pi}{2} = \sum_0^n \frac{n!}{(2n+1)!!}$

$R(n+1) = R(n) * \frac{n}{2n+1}, R(1) = 1$

$$sum = \pi = 2 * \sum_{a=1}^{\infty} R(n)$$

从公式中可以看出，我们只需要将每一个R(n)的值计算出，随后将结果加和乘二即可得到π的近似值。然而，如果使用这种办法去满足500位的精确度，我们就不能只计算前500位，而是要计算到600位以上。

在实际算法计算中，我们构造两个双链表分别用于计算每一位的R(n)和存储最终的近似数，记这两个链表为num与sum。

num表中首元节点记录了每一轮R(n)的个位，后续节点分别存储十分位、百分位、千分位……
sum表与num表各节点存储意义完全相同，但sum表存储的总体是最后满足精度需求的π的近似值。

利用指针循环移动先算分子作乘法，再算分母作除法，最后将结果存入结果链表，根据输入的n值来输出指定位数的π近似值。

> 基本的算法思路如下：
>
> - 定义两个双链表分别用于计算各R(n)的值和存储最后的π近似值，将链表中的各值置为0。
> - 在计算链表中，从尾部开始向表头计算分子作乘法，如果遇到进位，就将ret记录保存到下一节点在作乘法时加上。
> - 计算完乘法后，再从表头向表尾遍历节点作除法，如果遇到借位，就将ret记录保存到下一节点在作除法时加上ret*10，这样在每轮乘除结束之后，能够保证各节点所存储的R(n)每一位在进位和借位上的正确性。
> - 计算完乘法和除法后，将num当前的每一节点的数据域的值对应地加到sum表各节点中，并在计算时从高位到地位考虑进位，若存在进位，就将ret记录下来保存到下一节点加上。
> - 重复上述步骤，完成指定轮数的迭代（代码中设置了10000轮，即计算到R(10000)），以保整近似数的精确度要求。
> - 最后根据读入的n值来输出指定位数的π的近似值。

这里为了方便理解算法思路，将前两轮num表的迭代过程展示如下：

> - 初始设置num链表首元节点数据域值为2，这样能够省去结果乘2的过程，并且避免因为乘2导致的误差传播。
>   此时num链表数据域应该为 {Head, 2, 0, 0, ..., 0}，即R(0) = 2;
> - 在计算完第一轮乘法和除法后，根据迭代公式，此时num中存储的R(1)应当为0.6666...
>   实际num链表数据域为 {Head, 0, 6, 6, ..., 6}，即R(1) = 0.666...，与理论值一致。
> - 再迭代一轮，根据迭代公式，此时num中存储的R(2)应当为0.2666...
>   实际num链表数据域为 {Head, 0, 2, 6, 6, ..., 6}，即R(2) = 0.2666...，与理论值一致。

在迭代了10000次后，sum经过了10000次的num链表累加，已经十分逼近π的真值，达到了我们的精确度要求，此时只要按照输入的n值来输出指定位数的近似值即可。

这种大数乘法和大数除法的模拟手算程序实际上比较晦涩难懂，但只要理解实际操作时，每移动一次指针到下一节点，就是对该位小数做一次乘除，是一种以分配律的形式对每一位小数做相同计算的巧妙办法，在这种办法中最需要注意的就是进位和借位的判断以及对变量ret的传递设计。

3 算法实现

由于报告仅用于体现关键算法思想，故不再展示完整代码，仅对重要部分进行算法实现上的思路讲解。

3.1 归并数组

用单链表实现数组较为简单，因此不再赘述链表的创建与打印算法。

首先展示链表节点的数据结构：

```
typedef struct Node
{
    int data;
    struct Node *next;
} Node;
```

再展示实现归并的算法函数，讲解在代码块之后：

```
// merge sequence
Node *getMergedList(Node *a, Node *b)
{
    Node *result = (Node *)malloc(sizeof(Node));
    result->next = NULL;
    Node *current = result;

    a = a->next;
    b = b->next;
    while (a != NULL && b != NULL)
    {
        // printf("current a and b is: %d %d\n", a->data, b->data);
        if (a->data > b->data)
        {
            // put b into tail
            Node *newnode = (Node *)malloc(sizeof(Node));
            newnode->data = b->data;
            newnode->next = NULL;

            current->next = newnode;
            current = newnode;

            b = b->next;
        }
        else if (a->data < b->data)
        {
            // put a into tail
            Node *newnode = (Node *)malloc(sizeof(Node));
            newnode->data = a->data;
            newnode->next = NULL;

            current->next = newnode;
            current = newnode;

            a = a->next;
        }
        else
        {
```

```
            for (int i = 0; i < 2; i++)
            {
                Node *newnode = (Node *)malloc(sizeof(Node));
                newnode->data = a->data;
                newnode->next = NULL;

                current->next = newnode;
                current = newnode;
            }
            a = a->next;
            b = b->next;
        }
    }

    // Append the remaining elements of a or b
    while (a != NULL)
    {
        Node *newnode = (Node *)malloc(sizeof(Node));
        newnode->data = a->data;
        newnode->next = NULL;

        current->next = newnode;
        current = newnode;

        a = a->next;
    }

    while (b != NULL)
    {
        Node *newnode = (Node *)malloc(sizeof(Node));
        newnode->data = b->data;
        newnode->next = NULL;

        current->next = newnode;
        current = newnode;

        b = b->next;
    }

    return result;
}
```

尽管上述代码的注释很少，但是不难发现，代码具有很强的重复性，因此我们主要对循环和几个条件分值做讲解，将其单独提出如下：

```
while (a != NULL && b != NULL)
{
    // Add nodes with smaller data fields to the output
    if (a->data > b->data){
        ... ...
        b = b->next;
```

```
    }
    else if (a->data < b->data){
        ... ...
        a = a->next;
    }
    else{
        ... ...
        a = a->next;
        b = b->next;
    }

    // Append the remaining elements of a or b
    while (a != NULL){
        ... ...
        a = a->next;
    }
    while (b != NULL){
        ... ...
        b = b->next;
    }
}
```

由于我们的归并数组算法问题只运行一次，因此并没有定义新的指针用于遍历。

a表和b表是输入的待排序的两个升序链表，在遍历两表时比较其数据域，将较小值作为新节点加入输出链表，然后单独使较小值所在输入表指针后移，若两数据域相等，则执行两次新节点的加入，并将两表指针均后移。

在某一指针指向该表表尾时，说明另一表剩余元素均大于该表所有元素，此时仅需将另一表所有剩余节点全部按序加入输出链表即可。

算法结束。

3.2 高精度计算π值

同样的，链表的初始化、插入、打印算法不再赘述。

先展示链表节点的数据结构（实际同问题 3.1）：

```
typedef struct Node
{
    int data;
    struct Node *next, *pre;
} Node;
```

再展示反三角函数幂级数展开的计算与求取近似数的代码，讲解在代码块之后：

```
// 反三角函数幂级展开求Pi
void getPi(Node *num, Node *sum, int n)
{
```

```
    Node *p1 = num->next;
    Node *p2 = sum->next;
    p1->data = 2;
    p2->data = 2;

    // 将临时指针移到表尾作为表尾指针
    while (p1->next)
    {
        p1 = p1->next;
    }
    while (p2->next)
    {
        p2 = p2->next;
    }

    // 将计算用链表表尾指针赋给临时指针
    Node *numtail = p1;
    Node *sumtail = p2;

    // 计算，在num链表中完成
    int temp = 0;
    int ret = 0; // 进位数 || 借位数
    int t;       // 存储除数，即反三角函数展开的各项分母，在循环条件中完成迭代
    for (int i = 1, t = 3; i < 10000; i++, t += 2)
    {
        // 由于公式中存在n，可用i直接代替
        Node *p3 = numtail;
        ret = 0;
        // 从表尾向表头计算乘法
        while (p3->pre)
        {
            temp = p3->data * i + ret;
            p3->data = temp % 10;
            ret = temp / 10;
            p3 = p3->pre;
        }
        // 将进退位数置零，将p3从头结点挪回首元节点
        ret = 0;
        p3 = num->next;
        // 从表头向表尾计算除法
        while (p3->next)
        {
            temp = p3->data + ret * 10;
            ret = temp % t;
            p3->data = temp / t;
            p3 = p3->next;
        }
        // 本轮计算完成，将进退位数置零
        ret = 0;
        // 将计算结果从尾部插入sum表中
        Node *p4 = sumtail;
        while (p3 && p4)
        {
            temp = p3->data + p4->data + ret;
```

```
            ret = temp / 10;
            p4->data = temp % 10;
            p3 = p3->pre;
            p4 = p4->pre;
        }
    }
    printList(sum, n);
}
```

由于乘法涉及进位，除法涉及借位，故在进行大数乘法的模拟手算时，指针从表尾向表头移动，在模拟大数除法时，指针从表头向表尾移动。

首先定义了三个辅助变量：

```
// 计算，在num链表中完成
int temp = 0;
int ret = 0; // 进位数 || 借位数
int t;       // 存储除数，即反三角函数展开的各项分母，在循环条件中完成迭代
```

开始循环迭代，在num链表中进行指针遍历进行大数乘除的模拟手算，计算原理在 2.2 中已做详细介绍，此处不再赘述：

```
for (int i = 1, t = 3; i < 10000; i++, t += 2)
{
    // 由于公式中存在n，可用i直接代替
    Node *p3 = numtail;
    ret = 0;
    // 从表尾向表头计算乘法
    while (p3->pre)
    {
        temp = p3->data * i + ret;
        p3->data = temp % 10;
        ret = temp / 10;
        p3 = p3->pre;
    }
    // 将进退位数置零，将p3从头结点挪回首元节点
    ret = 0;
    p3 = num->next;
    // 从表头向表尾计算除法
    while (p3->next)
    {
        temp = p3->data + ret * 10;
        ret = temp % t;
        p3->data = temp / t;
        p3 = p3->next;
    }
    // 本轮计算完成，将进退位数置零
    ret = 0;
}
```

每进行一轮迭代，即计算完一轮乘除法后，num中存储的就是新的R(n)各位，将其对应加到sum表中即可。由于加法涉及进位，因此依旧是从表尾向表头计算。

```
for (int i = 1, t = 3; i < 10000; i++, t += 2)
{
    ... ...

    Node *p4 = sumtail;
    while (p3 && p4)
    {
        temp = p3->data + p4->data + ret;
        ret = temp / 10;
        p4->data = temp % 10;
        p3 = p3->pre;
        p4 = p4->pre;
    }
}
```

最后按n值输出sum表。

算法结束。

4 遇到的问题

归并数组问题十分简单，需要注意的问题是意识到某一表的指针有可能先遍历到表尾，此时若不将另一表其余元素加入输出链表，那么会造成元素的丢失。

高精度求π值问题最大的难点是将辅助Taylor展式在链表中体现出来，在计算num表中各节点的数据域在当轮迭代中的值时，进位和借位的体现如何实现也是十分困难的。