

**本科生实验报告**

|  |  |
| --- | --- |
| **实验课程** | 数据结构与程序设计 |
| **学院名称** | 核技术与自动化工程学院 |
| **专业名称** | 测控技术与仪器 |
| **学生姓名** | 李朝元 |
| **学生学号** | 202306010309 |
| **指导老师** | 朱杰 |
| **实验地点** | 6C802 |
| **实验成绩** |  |

**二〇二四年九月 —— 二〇二四年十二月**

**填写说明**

1. 适用于本科生所有的实验报告（印制实验报告册除外）；
2. 专业填写为专业全称，有专业方向的用小括号标明；
3. 格式要求：
4. 用A4纸双面打印（封面双面打印）或在A4大小纸上用蓝黑色水笔书写。
5. 打印排版：正文用宋体小四号，1.5倍行距，页边距采取默认形式（上下2.54cm，左右2.54cm，页眉1.5cm，页脚1.75cm）。字符间距为默认值（缩放100%，间距：标准）；页码用小五号字底端居中。
6. 具体要求：

**题目**（二号黑体居中）；

**摘要**（“摘要”二字用小二号黑体居中，隔行书写摘要的文字部分，小4号宋体）；

**关键词**（隔行顶格书写“关键词”三字，提炼3-5个关键词，用分号隔开，小4号黑体)；

正文部分采用三级标题；

**第1章** ××(小二号黑体居中，段前0.5行)

**1.1** ×××××小三号黑体×××××（段前、段后0.5行）

**1.1.1**小四号黑体（段前、段后0.5行）

**参考文献**（黑体小二号居中，段前0.5行），参考文献用五号宋体，参照《参考文献著录规则（GB/T 7714－2005）》。

**实验一 线性表的应用**

1. 实验目的
2. 掌握线性表的逻辑结构和存储结构特点；
3. 掌握线性表的基本操作，如建立、查找、插入和删除等。
4. 问题描述

智能家居系统创建一个家居环境参数表，包含“日期、时间、地点、温度、湿度”等信息。程序能够完成如下功能：

(1) 能够逐条输入信息，创建表；

(2) 能够显示表中的所有信息；

(3) 根据时间和地点进行查找，返回相关参数信息；

(4) 给定一条环境参数信息，按照日期和时间顺序插入到表中指定的位置；

(5) 删除指定日期的记录。

三、数据结构设计（选用的数据元素逻辑结构和存储结构实现形式说明）

（1）逻辑结构设计

采用链表来实现线性表的逻辑结构。每个节点包含环境参数数据（日期、时间、地点、温度和湿度）和指向下一个节点的指针。链表的特点使得在插入和删除操作时，能够高效地管理数据。

（2）存储结构设计

本实验选择链式存储结构来实现线性表。链式存储结构允许动态分配内存，适合频繁的插入和删除操作。每个节点包含数据域和指向下一个节点的指针，形成一条链表。

#define MAX\_LENGTH 4

typedef struct {

    char date[11]; // yyyy-mm-dd

    char time[6];  // hh:mm

    char addr[MAX\_LENGTH]; // 地址

    float temp;     // 温度

    float humi;     // 湿度

} home;

typedef struct HomeLNode {

home data;     // 结点的数据域

struct HomeLNode \*next; // 结点的指针域

} HomeLNode, \*HomeLinkList;

（3）存储结构形式说明

链表的存储结构是动态的，每个节点在运行时通过 malloc分配内存，节点之间通过指针连接。链表的头节点指向第一个有效数据节点，最后一个节点的指针指向 NULL，表示链表的结束。

四、算法设计

（1）算法列表（说明各个函数的名称，作用，完成什么操作）

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 名称 | 函数表示符 | 操作说明 |
| 1 | 创建链表 | CreateHomeList\_R() | 初始化链表，创建头节点并设置为空链表。 |
| 2 | 插入数据 | HomeListInsert() | 在指定位置插入新的环境参数信息。 |
| 3 | 显示所有信息 | DisplayList() | 遍历链表并打印所有节点的数据。 |
| 4 | 查找信息 | LocateElem() | 根据时间和地点查找对应的节点。 |
| 5 | 删除指定日期记录 | ListDeleteByDate() | 删除链表中指定日期的节点。 |

（2）各函数间调用关系（画出函数之间调用关系）

ListDeleteByDate()

删除指定日期记录

LocateElem()

查找信息

HomeListInsert()

插入数据

CreateHomeList\_R()

创建链表

DisplayList()

显示所有信息

（3）关键算法描述

**1.家居环境参数线性表创建算法**

本实验通过链式存储结构创建一个家居环境参数的线性表。该算法使用动态内存分配来生成链表的头节点，并初始化为空链表。创建函数 CreateHomeList\_R 的思路是分配内存并设置指针，确保链表的起始状态为有效。

代码如下：

void CreateHomeList\_R(HomeLinkList \*L) {

\*L = (HomeLinkList)malloc(sizeof(HomeLNode));

(\*L)->next = NULL; // 初始化为空链表

}

说明：该代码段中，首先通过 malloc为链表的头节点分配内存。将头节点的 next 指针设置为 NULL，表示当前链表为空，准备接收数据。

**2.数据插入算法**

插入算法允许用户在指定位置插入新的环境参数信息。该算法通过遍历链表找到插入位置，并创建新节点，将其插入到链表中。

代码如下：

Status HomeListInsert(HomeLinkList \*L, int i, home \*e) {

if (i < 1 || i > length + 1) return ERROR;

HomeLinkList p = \*L;

for (int j = 0; j < i - 1; j++) {

p = p->next; // 找到插入位置的前一个节点

}

HomeLinkList s = (HomeLinkList)malloc(sizeof(HomeLNode));

s->data = \*e; // 赋值

s->next = p->next; // 将新节点的next指向当前节点的next

p->next = s; // 将前一个节点的next指向新节点

length++;

return OK;

}

说明：该代码首先检查插入位置的合法性。然后通过循环找到插入位置的前一个节点 p，接着分配内存给新节点 s，并将新节点插入到链表中。更新链表的长度。

**3.查找算法**

查找算法根据时间和地点查找对应的节点，返回匹配的环境参数信息。

代码如下：

HomeLNode\* LocateElem(HomeLinkList L, char \*t, char \*a) {

HomeLinkList p = L->next;

while (p) {

if (strcmp(p->data.time, t) == 0 && strcmp(p->data.addr, a) == 0) {

return p; // 返回找到的节点

}

p = p->next;

}

return NULL; // 未找到

}

说明：该代码通过遍历链表，比较每个节点的时间和地点，找到匹配的节点并返回。如果未找到，返回 NULL。

**4.删除算法**

删除算法用于删除指定日期的记录，通过遍历链表找到匹配日期的节点，并调整指针以删除该节点。

代码如下：

Status ListDeleteByDate(HomeLinkList \*L, char \*dt) {

HomeLinkList p = \*L;

while (p->next) {

if (strcmp(p->next->data.date, dt) == 0) {

HomeLinkList q = p->next;

p->next = q->next; // 删除节点

free(q);

length--;

return OK;

}

p = p->next;

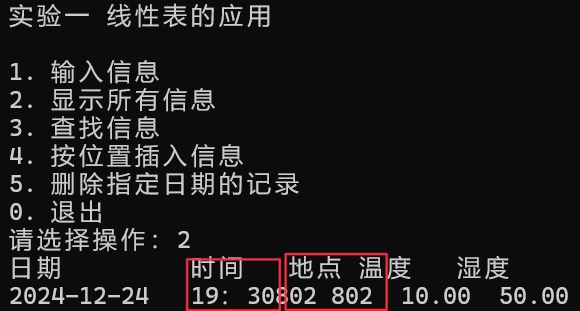
}

return ERROR;

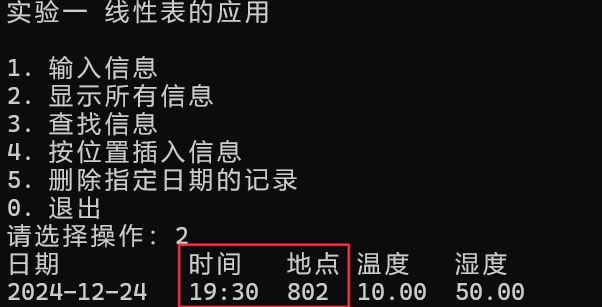
}

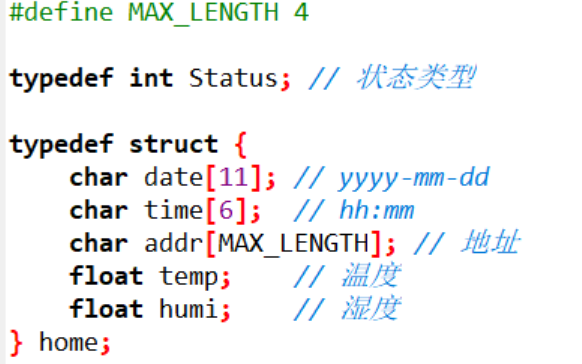
说明：该代码段通过遍历链表，找到日期匹配的节点，调整前一个节点的 next 指针，删除该节点并释放内存。更新链表的长度。

五、调试记录（调试过程中遇到的主要问题，是如何解决的，对设计和编码的回顾讨论和分析；改进设想等）



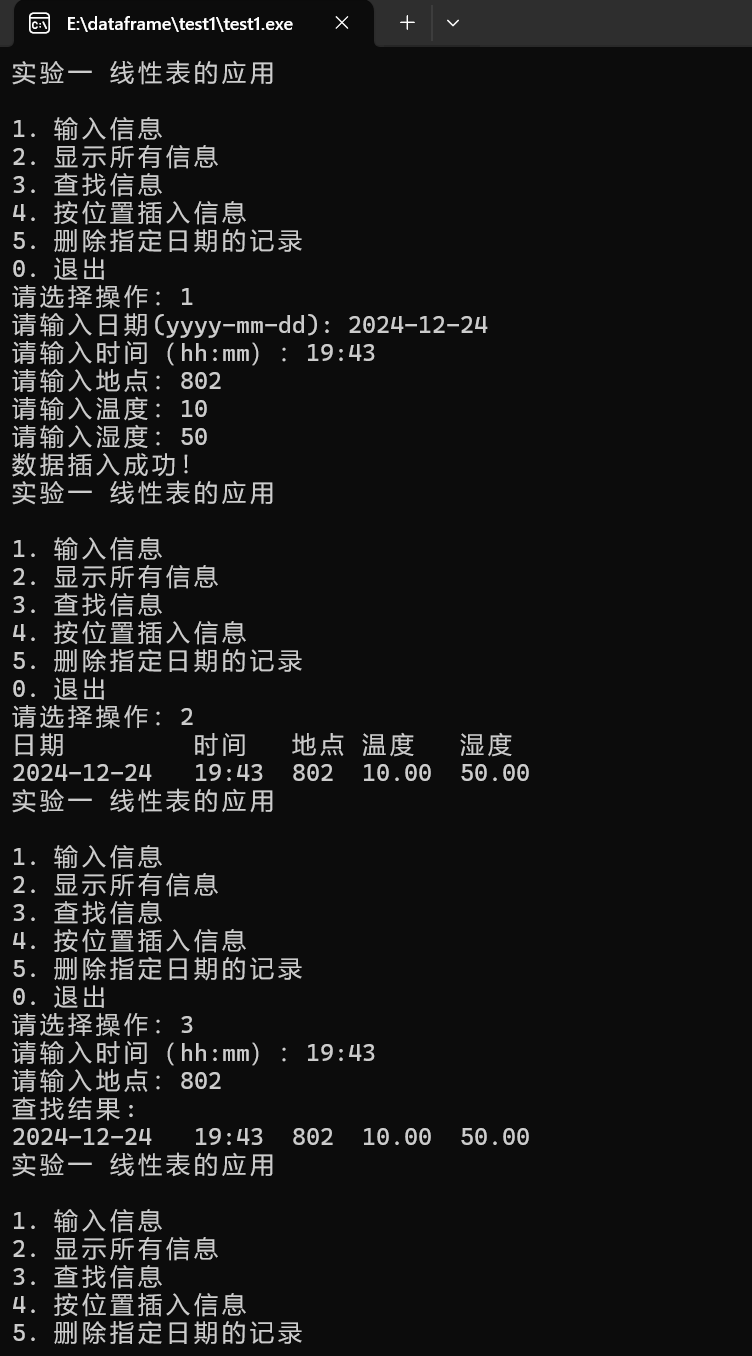
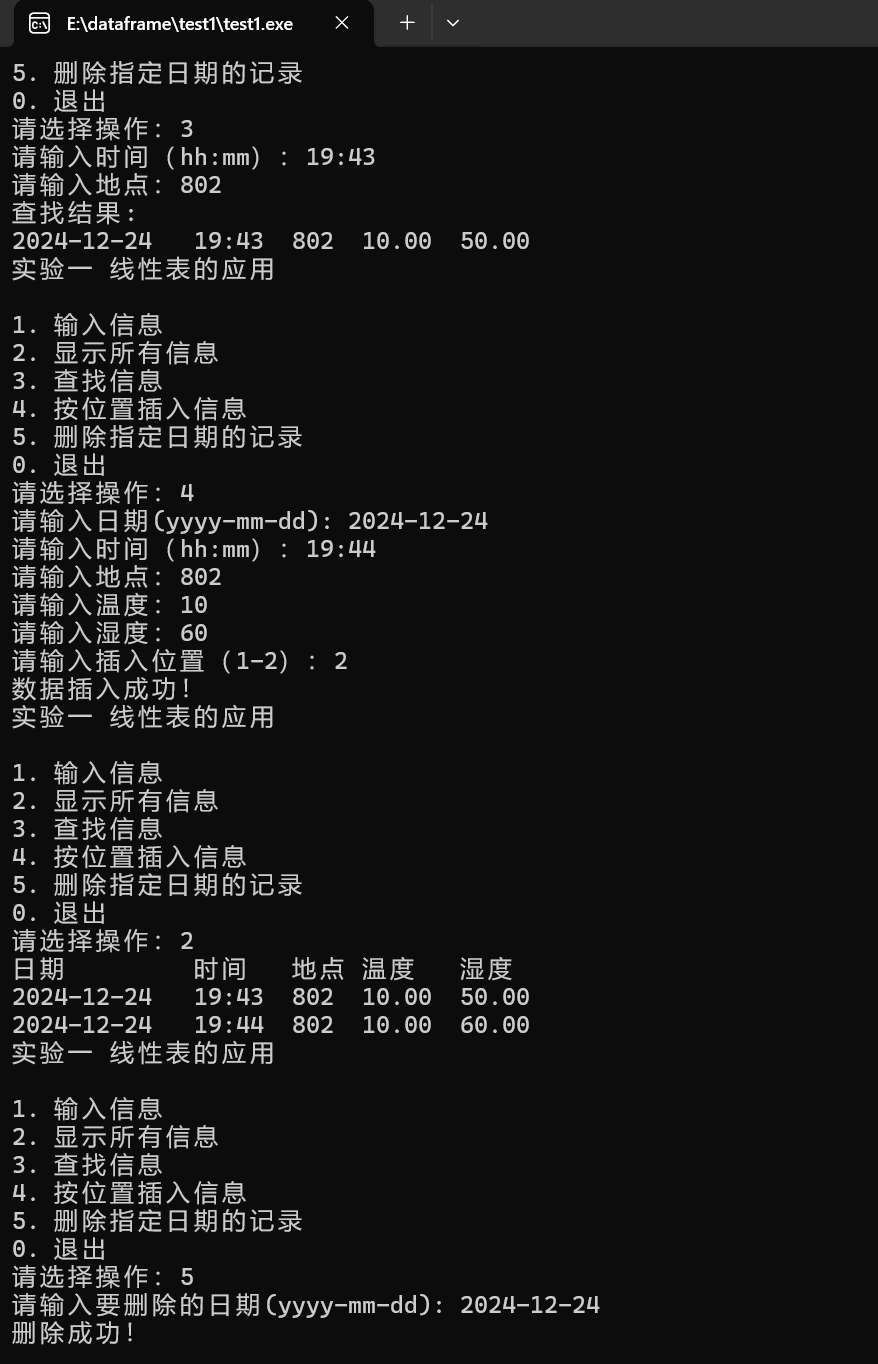
代码写完后运行第一遍时，没注意输入时间时“：”为中文的冒号，所以在显示字符串时出现了错误，并且影响到了旁边的地点显示，显示了两遍802，改进了下代码，第二遍运行显示就正常了





同时，我们在定义地址的宽度时可以根据我们所需要的宽度做一些修改，以避免地址宽度太宽，导致在显示地点时出现乱码，同时注意输入时的中英文的切换。

六、运行说明（列出测试结果，包括输入和输出。这里的测试数据应该完整和严格，最好多于示例中所列数据）

按指导书要求，测试输入，显示，查找，按位置插入，删除指定日期的记录功能，经多次测试验证，没有再出现调试遇到的问题，且功能实现一切正常。

**实验二 栈和队列的应用**

1. 实验目的：
2. 握栈和队列的逻辑结构及存储结构；
3. 运用栈和队列原理完成设计的内容。
4. 问题描述

1、完成数字十进制到八进制的转换。

输入示例：

请输入需转换的数的个数：

3

请输入需转换的数：

28，58，190

输出示例：

转换结果为：

1、 34

2、 72

3、 276

2、银行排队系统实现

功能要求：

（1） 客户进入排队系统；

（2） 客户离开；

（3） 查询当前客户前面还有几人；

（4） 查询截至目前总共办理多少客户。

输出要求：每进行一次操作后，输出当前排队成员情况。

三、数据结构设计（选用的数据逻辑结构和存储结构实现形式说明）

（1）逻辑结构设计

使用链式存储结构来实现栈和队列。栈采用后进先出的原则，队列采用先进先出的原则。栈的逻辑结构由一组节点组成，每个节点包含数据和指向下一个节点的指针。队列的逻辑结构则由两个指针（前指针和后指针）管理节点的插入和删除。

（2）存储结构设计

栈的存储结构：使用链式存储，每个节点包含一个数据域和指向下一个节点的指针。

typedef struct SNode {

int data; // 节点数据

struct SNode \*next; // 指向下一个节点的指针

} \*LinkStack;

队列的存储结构：使用顺序存储，定义一个数组和两个指针（前指针和后指针）来管理队列元素。

typedef struct {

Person \*base; // 存储元素的数组

int front; // 队头指针

int rear; // 队尾指针

} SqQueue;

（3）存储结构形式说明

**栈的存储结构：**采用链式存储结构。每个栈节点由一个数据域和一个指向下一个节点的指针组成。栈的顶端由一个指针指向最后插入的节点，从而实现后进先出的特性。

**队列的存储结构：**采用顺序存储结构。队列使用一个数组来存储元素，并用两个指针（前指针和后指针）来标识队头和队尾的位置，支持先进先出的操作。

四、算法设计

（1）算法列表（说明各个函数的名称，作用，完成什么操作）

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 名称 | 函数表示符 | 操作说明 |
| 1 | 栈初始化 | InitStack | 初始化栈，将栈顶指针设置为 NULL。 |
| 2 | 判断栈空 | StackEmpty | 判断栈是否为空，返回对应状态。 |
| 3 | 入栈操作 | Push | 将新元素压入栈顶，并更新栈顶指针。 |
| 4 | 出栈操作 | Pop | 从栈顶移除元素，返回该元素的值。 |
| 5 | 队列初始化 | InitQueue | 初始化队列，分配内存并设置头尾指针。 |
| 6 | 判断队空 | QueueEmpty | 判断队列是否为空，返回对应状态。 |
| 7 | 入队操作 | EnQueue | 在队列尾部插入新元素，并更新队尾指针。 |
| 8 | 出队操作 | DeQueue | 删除队头元素并返回该元素的值。 |
| 9 | 获取队头元素 | GetHead | 返回队头元素的值，如果队列为空则返回空值。 |
| 10 | 显示队列信息 | ShowInfo | 显示当前队列的状态，包括排队人数等信息。 |

（2）各函数间调用关系（画出函数之间调用关系）

Push

入栈操作

StackEmpty

InitStack

栈初始化

Pop

出栈操作

EnQueue(&custmers, custmer) 入队操作

初始化银行排队的队列

1.客户进入排队系统

输入选择

Showfront(custmers)

查询当前客户前面还有几人

DeQueue(&custmers, &custmer) 出队操作

ShowInfo

2.客户离开

3.查询当前

4.查询截至目前总共

Showsum(custmers)

查询截至目前总共办理多少客户

（3）算法描述

**1.栈初始化**

用于创建一个空栈，并将栈顶指针设置为 NULL。

Status InitStack(LinkStack \*S) {

\*S = NULL; // 设置栈为空

return OK;

}

说明：通过将栈指针设置为 NULL，实现栈的初始化，确保后续操作能够正常进行。

**2.入栈操作**

将新元素压入栈顶。

Status Push(LinkStack \*S, int e) {

LinkStack p = (LinkStack)malloc(sizeof(SNode)); // 分配新节点

if (!p) {

return OVERFLOW; // 存储分配失败

}

p->data = e; // 设置数据

p->next = \*S; // 新节点指向原栈顶

\*S = p; // 更新栈顶指针

return OK;

}

说明：该代码段为新节点分配内存并设置数据，更新指针，使新节点成为栈顶。

**3.出栈操作**

从栈顶移除元素，并返回该元素的值。

Status Pop(LinkStack \*S, int \*e) {

if (\*S == NULL) {

return ERROR; // 栈空

}

LinkStack p = \*S; // 保存栈顶节点

\*e = p->data; // 返回栈顶数据

\*S = (\*S)->next; // 更新栈顶指针

free(p); // 释放栈顶节点

return OK;

}

说明：检查栈是否为空，若不为空则返回栈顶数据，并更新栈顶指针。

**4.队列初始化和入队操作**

初始化队列并在队列尾部插入新元素。

int InitQueue(SqQueue \*Q) {

Q->base = (Person \*)malloc(MAXQSIZE \* sizeof(Person));

if (!Q->base)

exit(OVERFLOW); // 存储分配失败

Q->front = Q->rear = 0; // 头指针和尾指针置为零，队列为空

return OK;

}

int EnQueue(SqQueue \*Q, Person e) {

if ((Q->rear + 1) % MAXQSIZE == Q->front) // 判断队满

return ERROR;

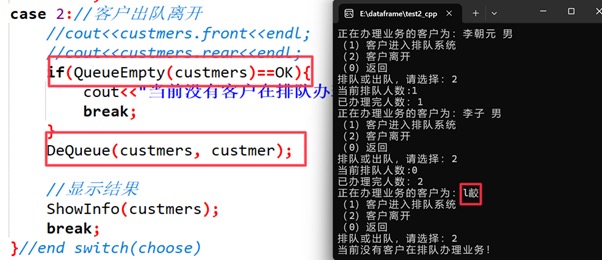
Q->base[Q->rear] = e; // 新元素插入队尾

Q->rear = (Q->rear + 1) % MAXQSIZE; // 队尾指针加1

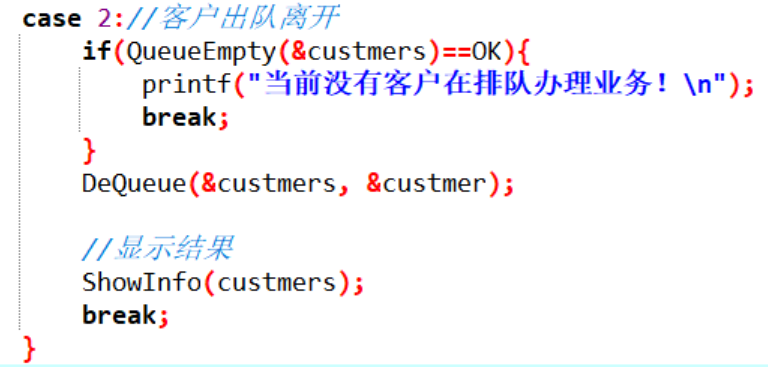
return OK;

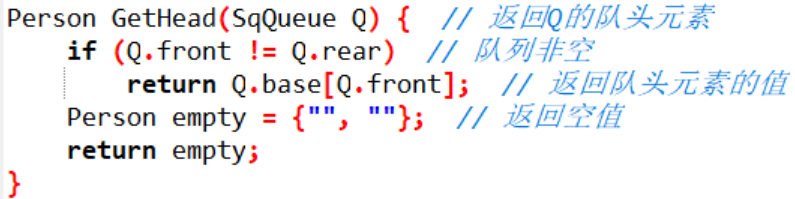
}

说明：初始化队列时分配内存并设置指针，入队操作时检查队列是否满，若未满则插入元素并更新指针。

五、调试记录（调试过程中遇到的主要问题，是如何解决的对设计和编码的回顾讨论和分析改进设想等）

在（2）客户离开这儿，逻辑出现了错，最初怀疑队列空判断函数出错，但并不是，输出front和endl就可以确定这个问题，在客户离开时，我们应该先删除再进行判断，否则我们需要连续退出两次，并且在第一次退出后，队列已经空了，这时候printf一个空的变量，就会出现乱码。将代码调整为下面的顺序就正常了。

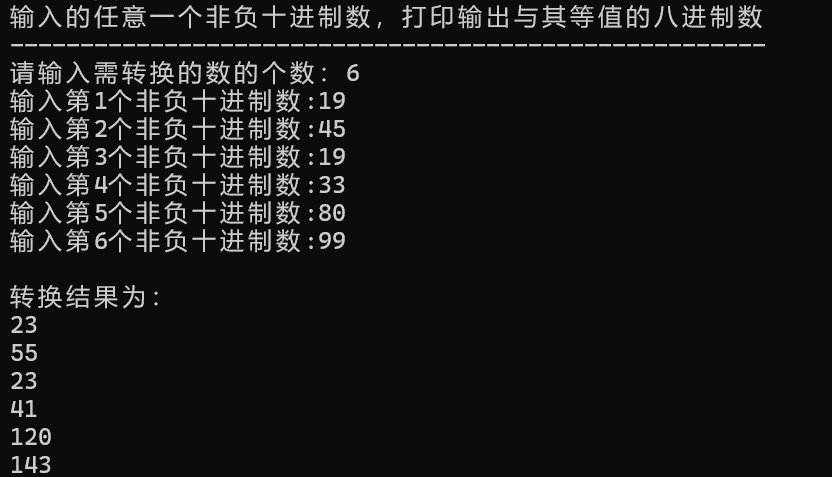
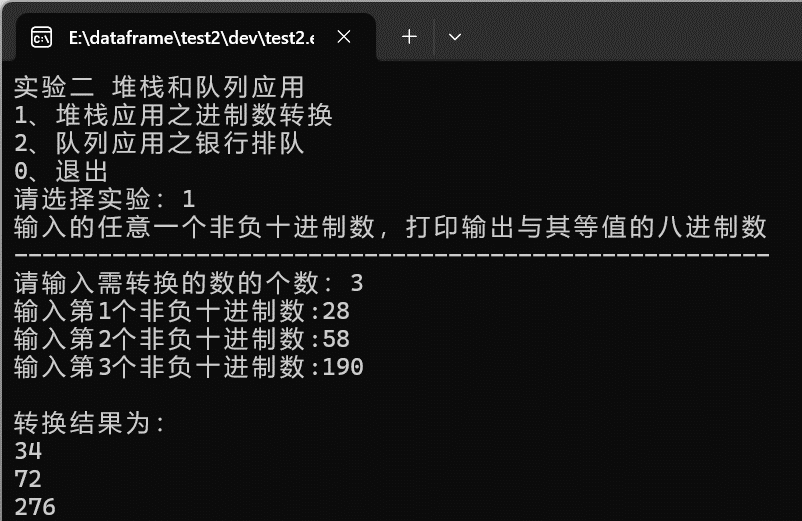




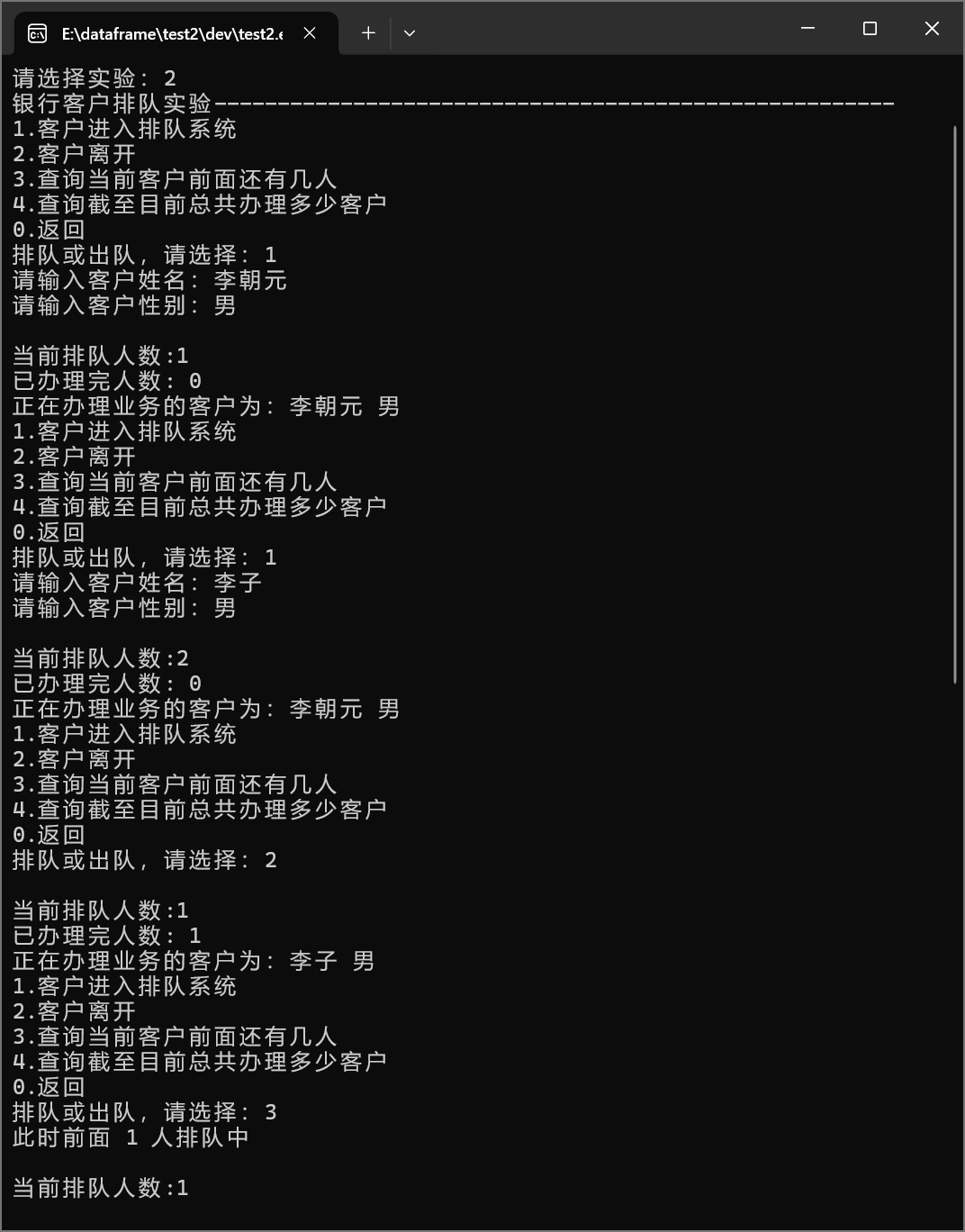
并且对于队列为空时，我们最好赋上{“ “,” ”},空的字符或字符串，避免输出乱码，增加代码的健壮性，以免出现bug

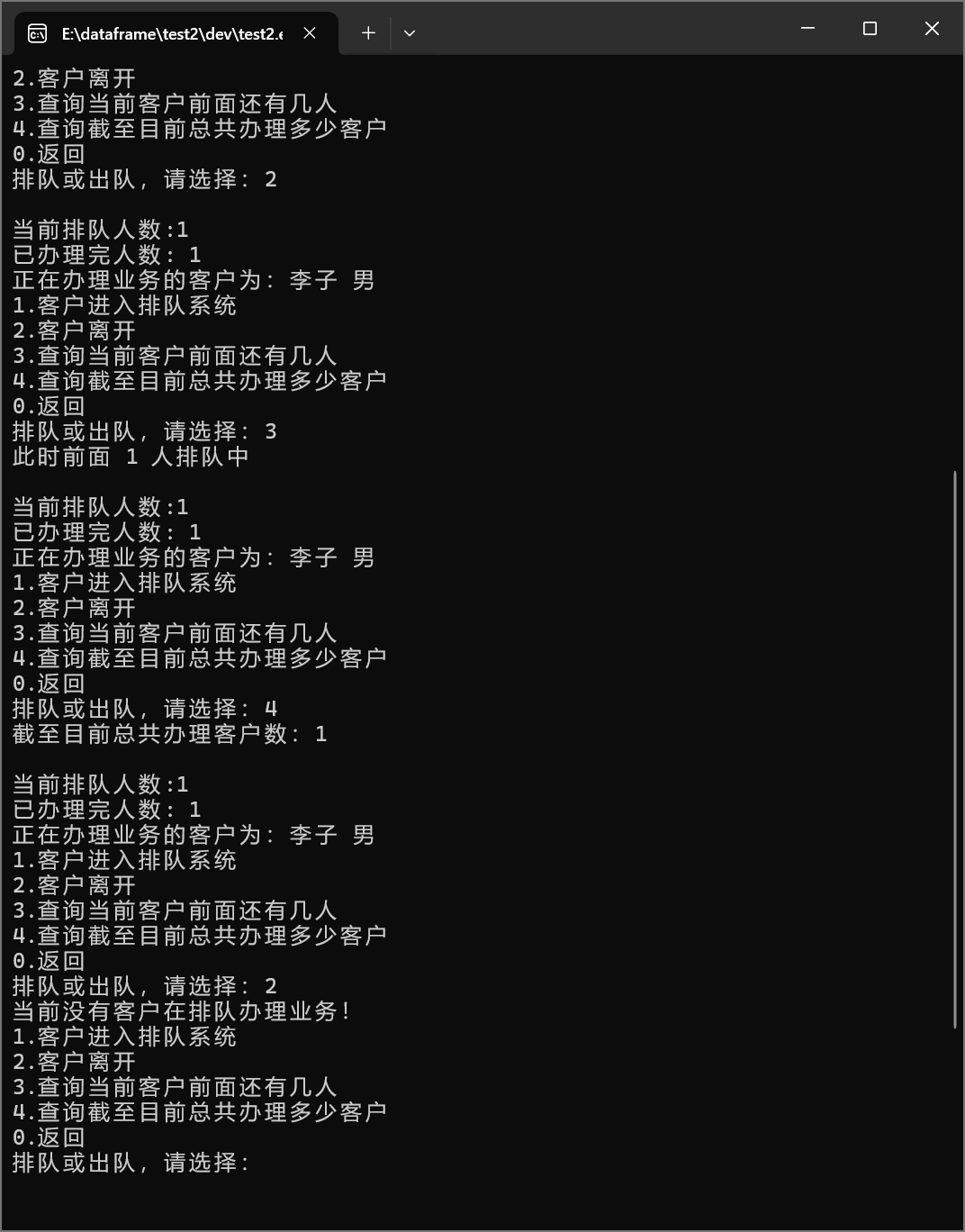
六、运行说明（列出测试结果，包括输入和输出。这里的测试数据应该完整和严格，最好多于示例中所列数据）

**1.堆栈应用之进制转换**



**2.队列应用之银行排队**





**实验三 树的应用**

1. 实验目的：
2. 掌握二叉树的定义和存储表示，掌握二叉树建立的算法；
3. 掌握二叉树的遍历（先序、中序、后序）算法。
4. 问题描述
5. 查找并绘制自己家族的族谱二叉树；
6. 族谱二叉树的建立（树的深度要>=4）；
7. 三种不同遍历算法遍历此二叉树；
8. 统计二叉树的深度，输出叶子结点的信息。

三、数据结构设计（选用的数据逻辑结构和存储结构实现形式说明）

（1）逻辑结构设计

采用二叉树的逻辑结构，节点由数据域和左右孩子指针组成。二叉树的每个节点最多有两个子节点，分别为左子节点和右子节点。

（2）存储结构设计

使用链式存储结构来表示二叉树。每个节点的结构体定义如下：

typedef struct BiNode {

char data[50]; // 节点数据域，可以存储字符串

struct BiNode \*lchild; // 左孩子指针

struct BiNode \*rchild; // 右孩子指针

} BiTNode, \*BiTree;

（3）存储结构形式说明

data[50]: 存储节点的数据，最多50个字符。

lchild: 指向左子树的指针，类型为 BiNode\*。

rchild: 指向右子树的指针，类型为 BiNode\*。

BiTNode: 表示二叉树节点的结构体，包含数据和左右孩子指针。

BiTree: 是指向 BiTNode 结构体的指针类型，用于方便表示树的根节点。

四、算法设计

（1）算法列表（说明各个函数的名称，作用，完成什么操作）

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 名称 | 函数表示符 | 操作说明 |
| 1 | 创建二叉树 | CreateBiTree | 递归输入节点值并构建二叉树。 |
| 2 | 中序遍历 | InOrderTraverse | 递归中序遍历二叉树并打印节点值。 |
| 3 | 先序遍历 | PreOrderTraverse | 递归先序遍历二叉树并打印节点值。 |
| 4 | 后序遍历 | PostOrderTraverse | 递归后序遍历二叉树并打印节点值。 |
| 5 | 计算树的深度 | Depth | 递归计算二叉树的深度。 |
| 6 | 统计叶子节点 | LeafNodes | 递归统计并打印所有叶子节点的信息。 |

（2）各函数间调用关系（画出函数之间调用关系）

CreateBiTree

创建二叉树

CreateBiTree

创建二叉树

CreateBiTree

创建二叉树

CreateBiTree

创建二叉树

CreateBiTree

创建二叉树

InOrderTraverse中序遍历

PostOrderTraverse后序遍历

PreOrderTraverse先序遍历

Depth

计算树的深度

LeafNodes

统计叶子节点

（3）算法描述

**1.创建二叉树算法**

该算法通过递归输入节点值构建二叉树。若输入为 "#"，则创建空树；否则创建新的节点，并递归创建其左、右子树。

void CreateBiTree(BiTree \*T) {

char ch[50];

printf("\n请输入节点的值：");

scanf("%s", ch);

if (strcmp(ch, "#") == 0) {

\*T = NULL; // 递归结束，建空树

} else {

\*T = (BiTree)malloc(sizeof(BiTNode));

strcpy((\*T)->data, ch); // 生成根节点

CreateBiTree(&(\*T)->lchild); // 递归创建左子树

CreateBiTree(&(\*T)->rchild); // 递归创建右子树

}

}

**2.中序遍历算法**

中序遍历算法递归地遍历二叉树，先访问左子树，再访问根节点，最后访问右子树。

void InOrderTraverse(BiTree T) {

if (T) {

InOrderTraverse(T->lchild);

printf("%s ", T->data);

InOrderTraverse(T->rchild);

}

**3.先序遍历算法**

先序遍历算法递归地访问当前节点，先访问根节点再访问左子树，最后访问右子树。

void PreOrderTraverse(BiTree T) {

if (T) {

printf("%s ", T->data);

PreOrderTraverse(T->lchild);

PreOrderTraverse(T->rchild);

}

}

**4.后序遍历算法**

后序遍历算法递归地访问左子树，再访问右子树，最后访问当前节点。

void PostOrderTraverse(BiTree T) {

if (T) {

PostOrderTraverse(T->lchild);

PostOrderTraverse(T->rchild);

printf("%s ", T->data);

}

}

**5.计算树的深度算法**

该算法递归计算二叉树的深度，空树的深度为0，非空树的深度为其左右子树深度的较大者加1。

int Depth(BiTree T) {

if (T == NULL) return 0; // 如果是空树，深度为0

int leftDepth = Depth(T->lchild); // 左子树深度

int rightDepth = Depth(T->rchild); // 右子树深度

return (leftDepth > rightDepth ? leftDepth : rightDepth) + 1; // 返回较大者加1

}

**6.统计叶子节点算法**

该算法递归遍历树，统计并打印所有叶子节点（没有子节点的节点）。

void LeafNodes(BiTree T) {

if (T) {

if (T->lchild == NULL && T->rchild == NULL) {

printf("叶子节点: %s\n", T->data);

}

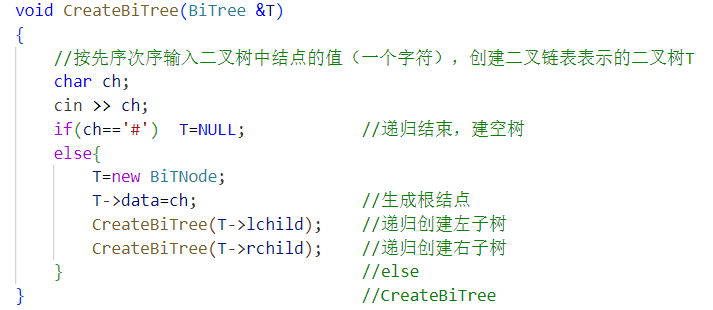
LeafNodes(T->lchild);

LeafNodes(T->rchild);

}

}

五、调试记录（调试过程中遇到的主要问题，是如何解决的，对设计和编码的回顾讨论和分析；改进设想等）



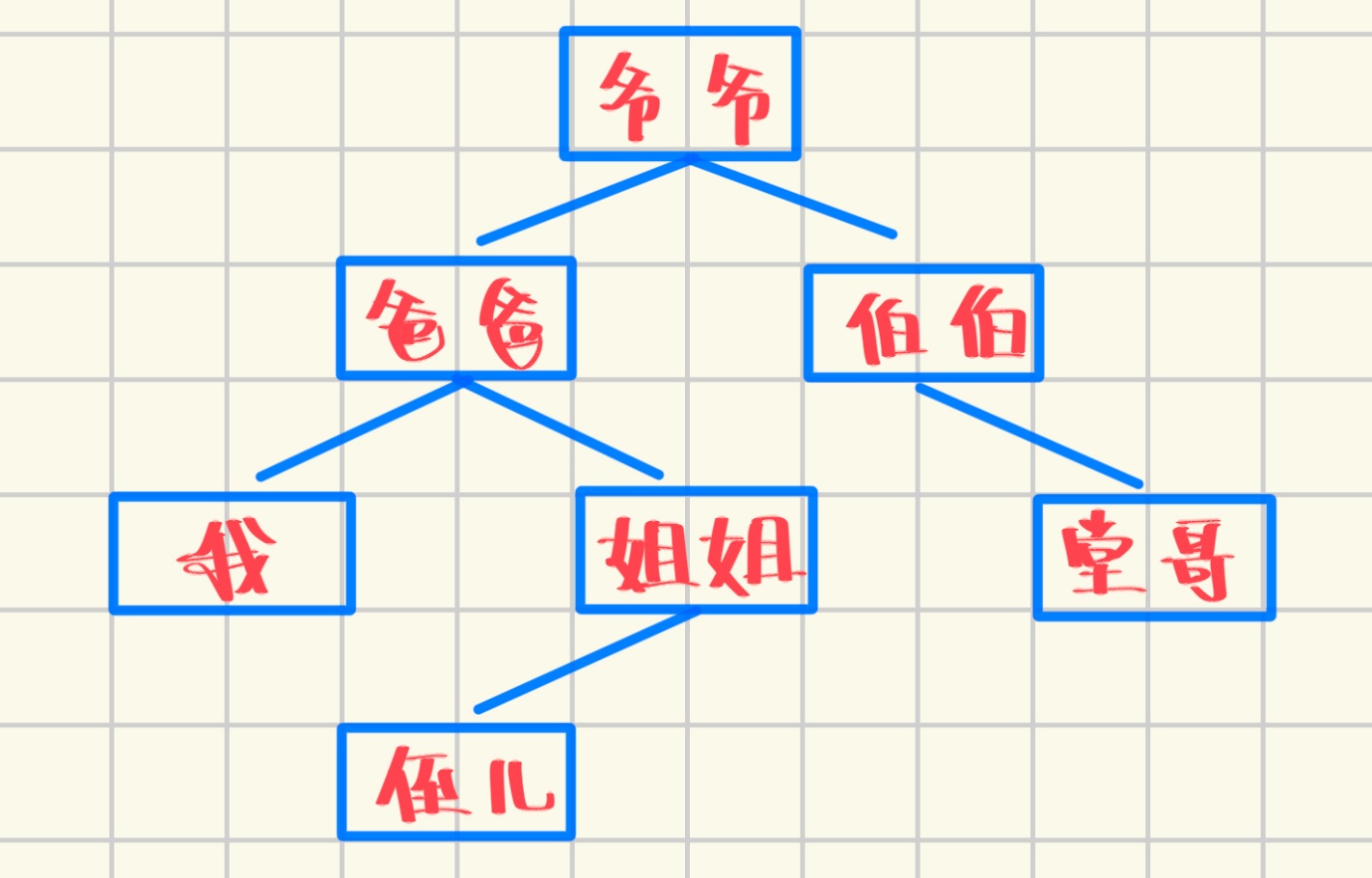
最初的代码在创建二叉树时只能一次性输入全部结点，一个字符作一个结点，这样的代码对于想要更改结点名称就特别不灵活，且自定义程度极低，所以我后面改成字符串的形式给结点命名，每次给一个结点命名，代码如下：

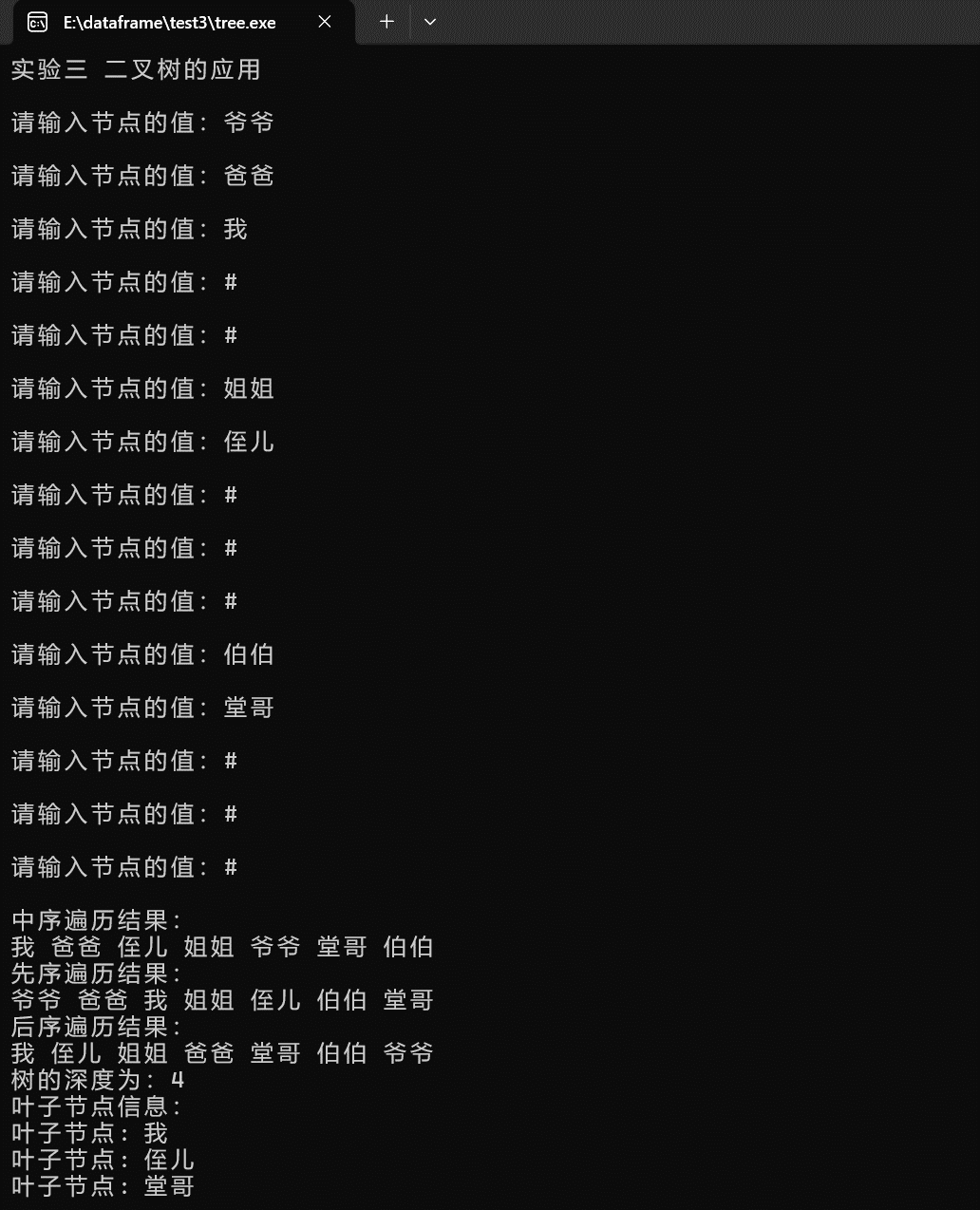


这样结点命名就可以不单单一个字符，可以是一串英文，汉字，或者英文加汉字

当然还可以继续优化，比如在每次输入时提示输入的是哪个结点的哪个子树，更清楚

六、运行说明（列出测试结果，包括输入和输出。这里的测试数据应该完整和严格，最好多于示例中所列数据）



当我们按照上图所示族谱关系图输入程序，便可验证程序的正确性

**实验四 图的应用**

1. 实验目的
2. 掌握图的基本概念和存储方法；
3. 掌握图的遍历算法，最短路径算法。
4. 问题描述
5. 绘制基于理工的地图网（结点不少于6），注：边的权值代表距离；实现网的创建；
6. 按照深度遍历和广度遍历算法输出结点信息；
7. 实现从西门到香樟的最短路径算法。

三、数据结构设计（选用的数据逻辑结构和存储结构实现形式说明）

（1）逻辑结构设计

使用邻接矩阵表示图，顶点用字符串表示，边的权值用整型表示。

（2）存储结构设计

使用邻接矩阵作为图的存储结构。邻接矩阵是一个二维数组，用于表示图中各个顶点之间的边和权值，适合用于稠密图的存储。每个元素表示两个顶点之间的边的权值，如果两个顶点之间没有边，则该元素的值为一个极大值（表示无穷大）。

存储结构的定义代码段

#define MaxInt 32767 // 表示极大值，即∞

#define MVNum 100 // 最大顶点数

#define MaxStrLen 20 // 顶点名称的最大长度

typedef char VerTexType[MaxStrLen]; // 顶点的数据类型

typedef int ArcType; // 边的权值类型

// 图的邻接矩阵结构

typedef struct {

VerTexType vexs[MVNum]; // 顶点表

ArcType arcs[MVNum][MVNum]; // 邻接矩阵

int vexnum, arcnum; // 当前图的顶点数和边数

} AMGraph;

（3）存储结构形式说明

VerTexType：定义顶点名称的类型为字符串，最大长度为20。

ArcType：定义边的权值类型为整型。

AMGraph结构体中包含：

vexs：存储所有顶点的名称。

arcs：邻接矩阵，用于存储各顶点之间的边的权值。

vexnum和arcnum：分别表示图中顶点的数量和边的数量。

四、算法设计

（1）算法列表（说明各个函数的名称，作用，完成什么操作）

|  |  |  |  |
| --- | --- | --- | --- |
| 序号 | 名称 | 函数表示符 | 操作说明 |
| 1 | 创建无向图 | CreateUDN | 创建邻接矩阵表示的无向图。 |
| 2 | 迪杰斯特拉算法 | ShortestPath\_DIJ | 计算从起始点到其他所有点的最短路径。 |
| 3 | 深度优先遍历 | DFS | 深度优先遍历图，并输出遍历结果。 |
| 4 | 广度优先遍历 | BFS | 广度优先遍历图，并输出遍历结果。 |
| 5 | 输出路径 | DisplayPath | 输出从起始点到终点的最短路径。 |

（2）各函数间调用关系（画出函数之间调用关系）

DFS

深度优先遍历00

DisplayPath输出路径

ShortestPath\_DIJ

迪杰斯特拉算法

CreateUDN

创建无向图

BFS

广度优先遍历

（3）算法描述

**1. 创建无向图**

void CreateUDN(AMGraph \*G) {

// 创建无向图的邻接矩阵表示

printf("请输入总顶点数，总边数，以空格隔开: ");

scanf("%d %d", &G->vexnum, &G->arcnum);

// 输入顶点名称

for (int i = 0; i < G->vexnum; ++i) {

printf("请输入第%d个点的名称: ", i + 1);

scanf("%s", G->vexs[i]);

}

// 初始化邻接矩阵

for (int i = 0; i < G->vexnum; ++i)

for (int j = 0; j < G->vexnum; ++j)

G->arcs[i][j] = MaxInt;

// 输入边及权值

for (int k = 0; k < G->arcnum; ++k) {

VerTexType v1, v2;

ArcType w;

printf("请输入第%d条边依附的顶点及权值: ", k + 1);

scanf("%s %s %d", v1, v2, &w);

int i = LocateVex(\*G, v1);

int j = LocateVex(\*G, v2);

G->arcs[i][j] = w;

G->arcs[j][i] = w;

}

}

**说明**：该函数负责创建无向图的邻接矩阵。首先，输入顶点数和边数，然后输入每个顶点的名称。接着，初始化邻接矩阵，将所有边的权值设为最大值表示无穷大。最后，输入每条边的两个顶点及其权值，并更新邻接矩阵。

**2. 迪杰斯特拉算法**

void ShortestPath\_DIJ(AMGraph G, int v0) {

// 初始化

for (int v = 0; v < G.vexnum; ++v) {

S[v] = false; // 标记顶点是否已被访问

D[v] = G.arcs[v0][v]; // 从起始点到各顶点的初始距离

Path[v] = (D[v] < MaxInt) ? v0 : -1; // 设置前驱节点

}

S[v0] = true; // 标记起始点已访问

D[v0] = 0; // 起始点到自身的距离为0

for (int i = 1; i < G.vexnum; ++i) {

int min = MaxInt, v;

for (int w = 0; w < G.vexnum; ++w)

if (!S[w] && D[w] < min) {

v = w;

min = D[w];

}

S[v] = true; // 访问顶点v

for (int w = 0; w < G.vexnum; ++w) {

if (!S[w] && (D[v] + G.arcs[v][w] < D[w])) {

D[w] = D[v] + G.arcs[v][w]; // 更新最短路径

Path[w] = v; // 更新前驱节点

}

}

}

}

**说明**：该函数实现了迪杰斯特拉算法。首先初始化每个顶点到起始点的距离和前驱节点。然后，通过循环不断寻找未访问顶点中距离最小的顶点，并更新其邻接顶点的最短路径和前驱节点。该算法的时间复杂度为O(V^2)，适用于稠密图。

**3. 深度优先遍历（DFS）**

void DFS(AMGraph G, int v, bool \*visited) {

visited[v] = true; // 标记当前顶点已访问

printf("%s ", G.vexs[v]); // 输出当前顶点

for (int w = 0; w < G.vexnum; ++w) {

if (G.arcs[v][w] < MaxInt && !visited[w]) {

DFS(G, w, visited); // 递归访问未访问的邻接顶点

}

}

}

**说明**：该函数实现深度优先遍历。通过递归访问所有未访问的邻接顶点，直到所有可达的顶点都被访问。

**4. 广度优先遍历（BFS）**

void BFS(AMGraph G, int start) {

bool visited[MVNum] = {false}; // 初始化访问标记

int queue[MVNum], front = 0, rear = 0; // 初始化队列

visited[start] = true;

queue[rear++] = start; // 添加起始顶点

printf("%s ", G.vexs[start]);

while (front < rear) {

int v = queue[front++]; // 出队

for (int w = 0; w < G.vexnum; ++w) {

if (G.arcs[v][w] < MaxInt && !visited[w]) {

visited[w] = true; // 标记已访问

queue[rear++] = w; // 入队

printf("%s ", G.vexs[w]); // 输出当前顶点

}

}

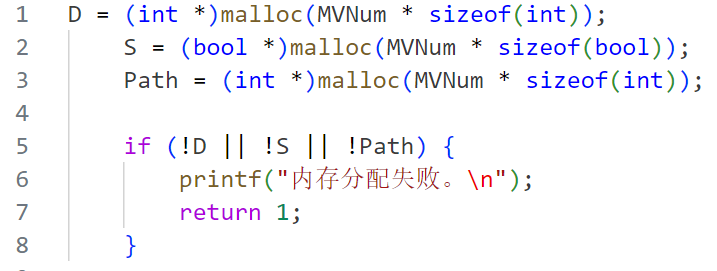
}

}

**说明**：该函数实现广度优先遍历。使用队列依次访问每个顶点，并记录已访问的状态，从而实现层次遍历。

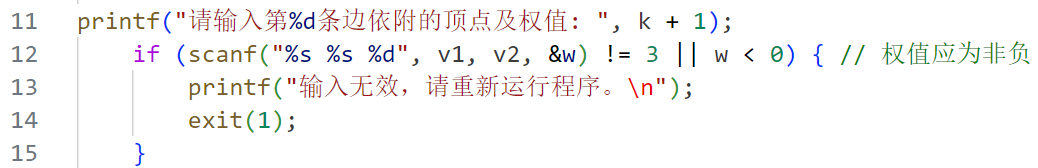
五、调试记录（调试过程中遇到的主要问题，是如何解决的，对设计和编码的回顾讨论和分析；改进设想等）

1. **内存管理问题**



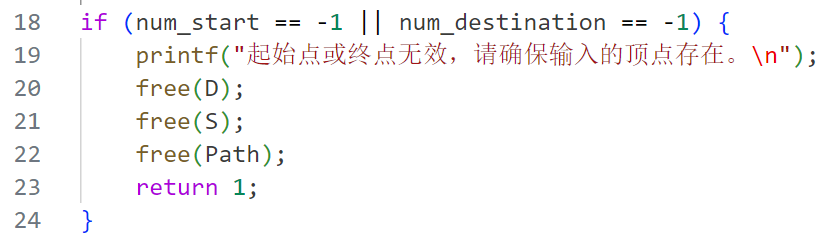
在动态分配内存后，未检查是否分配成功，可能导致后续操作出现未定义行为。在每次 malloc 后添加检查，确保内存分配成功。

**2. 输入验证问题**

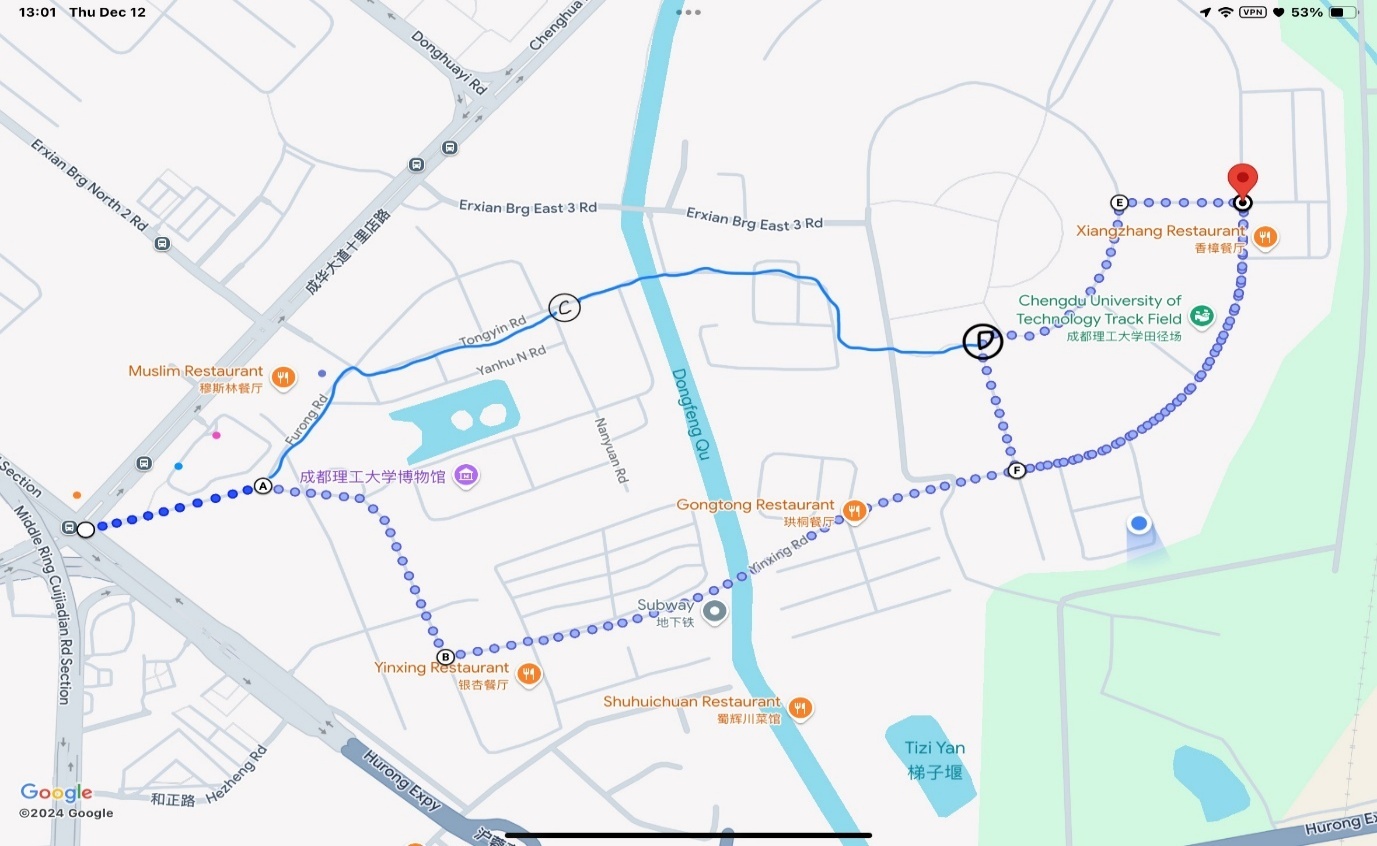


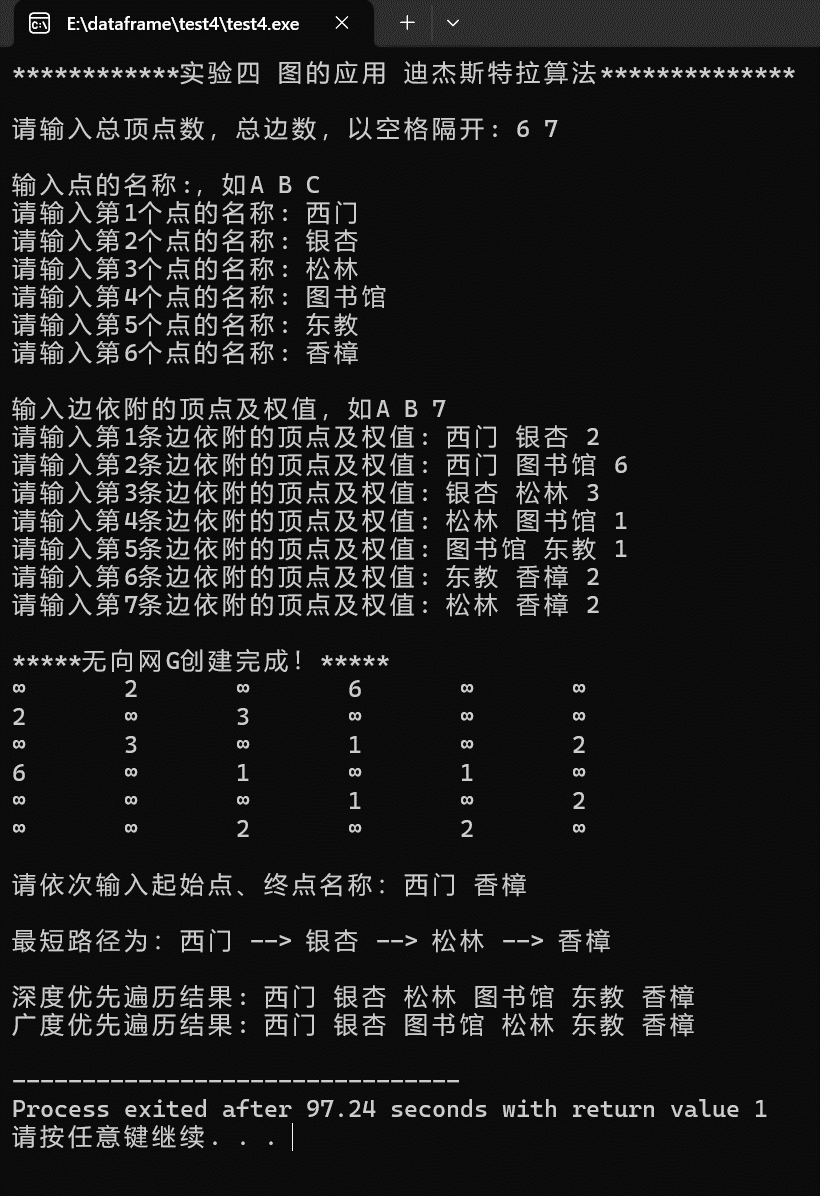
在输入顶点名称和边的权值时，代码未对输入格式进行验证，用户输入错误可能导致程序崩溃。通过增加输入验证，确保输入符合预期格式。例如，检查顶点数量和边的数量的有效性。

**3.路径显示问题**



DisplayPath 函数在处理路径时未考虑起点，起点会重复显示。 通过修改 DisplayPath 函数，增加判断，确保不会重复显示起点。

六、运行说明（列出测试结果，包括输入和输出。这里的测试数据应该完整和严格，最好多于示例中所列数据）



根据上面的地图，输进我们的程序，先是创建一个无向网G，根据输出的可以验证程序的正确性，接着输入我们的起点与终点，根据迪杰斯特拉算法得出我们的最短路径，最后是深度、广度优先遍历，经验证，符合我们所创建的理工大学地图。

|  |  |
| --- | --- |
| **学生实验 心得** | 在本学期的数据结构与程序设计课程中，通过四个实验的实践，我对数据结构的理论知识和实际应用有了更深入的理解。线性表的应用实验让我掌握了基本操作，如插入、删除和查找。  栈和队列的实验让我学习了这两种数据结构的逻辑结构与存储结构，明白了它们在实际应用中的关键作用。例如，在银行排队系统中，通过队列来管理客户的进出。十进制到八进制的转换让我体验到算法设计的乐趣，巩固了我的理论基础。  二叉树的实验中，通过实现不同的遍历算法，我认识到数据结构的灵活性和多样性。这些实验让我体会到，数据结构不仅是存储数据，更是对数据关系的有效组织。  在实验过程中，我遇到了许多挑战，尤其是链表的指针操作，这让我深刻认识到细心和耐心的重要性。通过反复调试，我提高了对指针操作的敏感度和解决问题的能力。  **特别感谢**老师在整个学习过程中的帮助与支持。老师耐心解答我的疑惑，提供了宝贵的建议，使我在每个实验中不断进步。老师的鼓励让我在面对困难时更加自信。  未来，我希望能更加深入地探索图的应用，特别是在最短路径算法方面。同时，我也计划继续提升我的编程能力，特别是在算法设计与优化方面。这门课程不仅让我掌握了数据结构的基本知识，也培养了我的逻辑思维能力，为今后的学习和工作打下了坚实的基础。  学生（签名）：李朝元  2024 年 12月 22日 |
| **指导**  **教师**  **评语** | 成绩评定：  指导教师（签名）：  年 月 日 |