实验一 线性表的应用：稀疏一元多项式运算器

崔士强 PB22151743

1 问题描述

1.1 设计目标

本实验的设计目标是创建一个稀疏一元多项式运算器。这个运算器需要实现对稀疏一元多项式的基本操作，包括创建、输出、求和、求差、求值、销毁、清空和修改多项式，另外实现多项式的微分，定积分，不定积分功能。

1.2 输入及输出

每次输入多项式时，逐个输入各项的系数以及指数，程序将这些项按升序排列。经过运算后的多项式按要求存放在列表中，通过函数`void PrintPolyn(Polynomial P)` 输出到屏幕上

2 算法描述

2.1 数据结构描述

本程序使用链表存储多项式，用一个数组存放所有多项式：

```
typedef struct term{        // 多项式的项
    float  coef;              // 系数
    int    exp;               // 指数
}term, ElemType;

typedef struct LNode{       // 结点类型
    ElemType      data;
    struct LNode  *next;
}*Link, *Position;

typedef struct LinkList{    // 链表类型
    Link   head, tail;        // 分别指向线性链表的头结点和最后一个结点
    int    len;               // 指示线性链表中数据元素的个数
}LinkList;

typedef LinkList polynomial; //用带表头结点的有序链表表示多项式
```

2.2 程序结构描述

函数原型及功能说明如下：

```
void CreatePolyn(polynomial &P, int m );
  // 输入m项的系数和指数，建立表示一元多项式的有序链表P
void DestroyPolyn(polynomial &P);
  // 销毁一元多项式P
void PrintPolyn(polynomial P);
  // 打印输出一元多项式P
void ClearPolyn(polynomial p);
  // 将一元多项式P置空
int PolynLength(polynomial P);
  // 返回一元多项式P中的项数
void AddPolyn(polynomial &Pa, polynomial &Pb);
  // 完成多项式相加运算，即：Pa=Pa+Pb，并销毁一元多项式Pb
void SubstractPolyn(polynomial &Pa, polynomial &Pb);
  // 完成多项式相减运算，即：Pa=Pa-Pb，并销毁一元多项式Pb
float EvaluatePolyn(polynomial P, float f);
  // 算出一元多项式P(x)在x=f处的值
void DiffPolyn(polynomial &P);
  // 求一元多项式P的1阶导数
void Integral(polynomial &p);
  // 求一元多项式P的不定积分
void IntPrintPolyn(polynomial P);
  // 打印输出P的不定积分
float IntEvalPolyn(polynomial P, float f);
  // 算出一元多项式P的不定积分在x=f处的值
float DefIntegral(polynomial P, float a, float b);
  // 求一元多项式P在[a, b]上的定积分
```

3 调试分析

3.1 测试数据

选择含正、负、零次项的多项式进行测试，以保证正确性

3.2 问题及解决方法

在测试过程中发现对于 -1 次项的积分，由于其原函数为对数形式，需要单独处理.

解决方法：输出不定积分时，对每一项判断是否是零次项，如果某一项是零次项则输出对数形式。求定积分时同理.

4 算法的时空分析

各项功能的时间复杂度如下所示：

操作	相加/相减	求值	微分	不定积分	定积分
时间复杂度	$O(N+M)$	$O(N)$	$O(N)$	$O(N)$	$O(N)$

5 测试结果及分析

$$P(x) = x^{-1} + 1 + x + 2x^2 + 3x^3 + 4x^4 + 5x^5$$

$$Q(x) = 5x + 4x^2$$

对两个多项式执行以下操作（每次操作后复原）：

1. $P(x) + Q(x)$，并输出序号为 1 的多项式
2. $P(x) - Q(x)$，并输出序号为 1 的多项式
3. 计算 $P(x)$ 在 $x = 4$ 处的值
4. 计算 $P(x)$ 的 3 阶导数
5. 计算 $P(x)$ 的不定积分，并求其在 $x = 1$ 处的值
6. 计算 $P(x)$ 从 -1 到 4 的定积分
7. 将 $P(x)$ 置空，并输出序号为 1 的多项式
8. 将 $P(x)$ 销毁，并输出序号为 1 的多项式

程序输出的结果均符合预期.

6 实验体会和收获

在编写链表相关的函数并进行运用的过程中熟练掌握了链表的相关结构及算法，在编写多项式相关操作算法的过程中增强了对指针操作的把握程度。

实验二 栈与队列的应用——括号配对检验

崔士强 PB22151743

1 问题描述

假设一个表达式有英文字母（大、小写）、数字、四则运算符 (+ − ∗ /) 和左右小括号、中括号、大括号构成，以“@”作为表达式的结束符。本程序检查表达式中的左右大中小括号是否匹配，若匹配，则返回”YES”; 否则返回”NO”。

输入文件中第一行是表达式数目 N，之后是需要进行括号配对检测的 N 个表达式。N 行输出分别对应输入的 N 行表达式，每行都为”YES” 或”NO”

2 算法描述

2.1 数据结构描述

本程序利用栈存储符号，读取到左括号则进栈，读取到右括号则出栈，相关类型定义如下：

```
typedef char SElemType;

typedef struct {
    char *base;
    char *top;
    int stacksize;
} SqStack;
```

2.2 程序结构描述

栈的相关函数声明如下：

```
// 栈操作函数声明
Status InitStack(SqStack &S);
    // 构造一个空栈S
Status DestroyStack(SqStack &S);
    // 销毁栈S，S不再存在
Status ClearStack(SqStack &S);
    // 把S置为空栈
Status StackEmpty(SqStack S);
    // 若栈S为空栈，则返回TRUE，否则返回FALSE
int StackLength(SqStack S);
```

```
11      // 返回S的元素个数，即栈的长度
12  Status GetTop(SqStack S, SElemType &e);
13      // 若栈不空，则用e返回S的栈顶元素，并返回OK；否则返回ERROR
14  Status Push(SqStack &S, SElemType e);
15      // 插入元素e为新的栈顶元素
16  Status Pop(SqStack &S, SElemType &e);
17      // 若栈不空，则删除S的栈顶元素，用e返回其值，并返回OK；否则返回ERROR
18  Status StackTraverse(SqStack S, Status (*visit)(SElemType &e));
19      // 从栈底到栈顶依次对栈中每个元素调用函数visit()。一旦visit()失败，则操作失败
```

3 调试分析

3.1 测试数据

选取包含各种错误（如配对不正确，优先级不正确）的表达式进行测试

3.2 问题及解决方法

调试中发现程序并不能正确地对括号优先级进行检验，每次对左括号进栈时检查栈顶元素可以解决这个问题。

4 算法的时空分析

从以下几个方面分析：

1. 文件读取和输出：文件的读取和输出操作在这里是线性的，因为它们依次处理文件中的每一行。但这通常不是主要的时间复杂度来源，除非文件非常大。
2. 外部循环：外部 for 循环迭代的次数等于文件中的行数 LineNum。
3. 内部循环：内部 for 循环遍历每一行中的字符直到遇到'@' 字符。假设每行的平均字符数为 M，则内部循环的复杂度是 $O(M)$。
4. 栈操作：栈操作（如 Push, Pop, GetTop, StackEmpty）是常数时间操作，即 $O(1)$。

因此，总的时间复杂度是外部循环和内部循环的乘积，即 $O(LineNum \times M)$。这意味着算法的执行时间随着输入文件的行数和每行的平均长度的增加而线性增长。

5 测试结果及分析

测试用数据如下所示：

```
1  6
2  3*(4+5)-{6/[7*(8-9)]}@
3  {[1+2*(3-4)]/5}+6-7@
4  1+[2*(3-4]}/5@
5  6*{7+[8-(9*10]}@
```

```
8/{9+(10-[11*12])}@
{[(2+3*4]/(5-6)+7}@
```

输出结果:

```
Yes
Yes
No
No
No
No
```

全部正确。

6 实验体会和收获

掌握了栈的相关结构和算法，对栈的应用场景以及使用栈的原因有了更深刻的理解

实验二 栈与队列的应用——银行业务模拟

崔士强 PB22151743

1 问题描述

1.1 设计目标

客户业务分为两种：

1. 申请从银行得到一笔资金，即取款或借款
2. 向银行投入一笔资金，即存款或还款

银行有两个服务窗口，相应地有两个队列。客户到达银行后先排第一个队，处理每个客户业务时，如果属于第一种，且申请额超出银行现存资金总额而得不到满足，则立刻排入第二个队等候直至满足时才离开银行; 否则业务处理完后立刻离开银行, 每接待完一个第二种业务的客户，则顺序检查和处理 (如果可能) 第二个队列中的客户，对能满足的申请者予以满足, 不能满足者重新排到第二个队列的队尾。注意，在此检查过程中，一旦银行资金总额少于或等于刚才第一个队列中最后一个客户 (第二种业务) 被接待之前的数额，或者本次已将第二个队列检查或处理了一遍，就停止检查 (因为此时已不可能还有能满足者) 转而继续接待第一个队列的客户。任何时刻都只开一个窗口。假设检查不需要时间，营业时间结束时所有客户立即离开银行。

本程序通过模拟方法求出客户在银行内逗留的平均时间

1.2 输入输出

第一行输入四个数 `N`、`total`、`close_time`、`average_time`，分别表示来银行的总人数、银行开始营业时拥有的款额、今天预计的营业时长和客户交易时长之后的 N 行每行输入两个数 `a`、`b`，第一个数 `a` 为客户办理的款额，用负值和正值分别表示第一类和第二类业务。第二个数 `b` 为客户来到银行的时间

2 算法描述

2.1 数据结构描述

本程序设置两个类`Customer`和`Bank`，用队列处理顾客相关状态：

```
1  typedef Customer QElemType;
2
3  typedef int Status;
4
```

```
typedef struct QNode{
    QElemType     data;
    struct QNode  *next;
}QNode, *QueuePtr;
```

2.2 程序结构描述

程序中用到的两个类的声明如下所示：

```
class Customer
{
    public:
        int money;
        int arrivalTime;
        int waitTime;
        int leaveTime;
        int order;
};
class Bank
{
    public:
        int money;
        int closeTime;
        int serviceTime;
        int waitTimeList[100];
        int pNumber;
        int clock;
        int open;
        int avgWaitTime;
        LinkQueue Queue1, Queue2;
        Status deal(LinkQueue& queue, Customer &customer);
            // 处理queue中一位客户customer的业务
        void checkQueue1();
            // 对Queue1进行一次处理
        void checkQueue2(int benchmark);
            // 对Queue2进行检查
        void close();
            // 结束业务
};
```

3 调试分析

3.1 测试数据

测试过程中选取会导致银行持有资金无法满足需求的测试样例，以此测试第二个队列处理的正确性。另外选取时间累积超过营业时间的样例以测试结束业务的过程是否正确执行。

4 算法的时空分析

1. 初始化和输入操作：初始化银行和顾客的操作是常数时间操作 $O(1)$。然后，根据银行的顾客人数 N_p 读取顾客信息，这部分的时间复杂度是 $O(N_p)$.

2. 主循环：`while(bank.clock < bank.closeTime)`循环的次数取决于银行的关闭时间、顾客的到达时间以及服务时间。在这个循环中，有几个关键操作：

   `EnQueue(bank.Queue1, customer[i])`：队列的入队操作是常数时间 $O(1)$。

   `bank.checkQueue1()`：这个方法可能调用 `checkQueue2`。`checkQueue2` 的时间复杂度是 $O(n)$，其中 n 是队列 `Queue2` 的长度。因此，`checkQueue1` 的时间复杂度也可能受到 `Queue2` 长度的影响。

3. 输出和平均等待时间计算：输出每个顾客的等待时间并计算平均等待时间的循环时间复杂度是 $O(N_p)$。

考虑以上各点，`main` 函数的总体时间复杂度依赖于几个关键因素：顾客的数量 N_p 和队列 `Queue2` 的最大长度。假设 `Queue2` 的最大长度为 m，则 `checkQueue1` 在最坏情况下的时间复杂度是 $O(m)$。由于 `checkQueue1` 在主循环中被调用，因此 `main` 函数的总体时间复杂度为 $O(N_p \times m)$.

5 测试结果及分析

测试数据如下所示：

```
4 10000 600 10
-2000 0
1000 10
-10000 30
2000 50
```

输出结果：

```
0
0
30
0
7
```

符合预期.

6 实验体会和收获

通过本实验，熟悉了关于队列的结构、操作、算法以及适用场景.

实验三 二叉树的应用

崔士强 PB22151743

1 问题描述

程序需要完成的操作包括：

1. 输入电文字符串
2. 统计电文字符集和每种字符在电文中出现的次数
3. 构建 huffman 树
4. 产生每种字符的 huffman 编码
5. 将电文串翻译成比特流
6. 对电文比特流进行解码

输出信息如下所示：

原始字符	编码后字符
aaabbbcccddd...	一串乱码
频率信息	编码信息
a 5000	a 1100
b 20000	b 111
c 8000	c 1101
d 4000	d 0
e 27000	e 10
.....	.....

2 算法描述

2.1 数据结构描述

本程序所用到的 Huffman 树存储结构如下所示：

```
typedef struct {
  int weight;
  int parent, lchild, rchild;
} HTNode, *HuffmanTree;

typedef char **HuffmanCode;
```

2.2 程序结构描述

程序中的函数声明如下:

```
void Select(HuffmanTree HT, int n, int &s1, int &s2);
  // 在HT[1..n]中选择parent为0且weight最小的两个结点，其序号分别为s1和s2

void HuffmanCoding(HuffmanTree &HT, HuffmanCode &HC, int *weight, int n);
  // 用n个字符的权值weight构造Huffman树HT,并求出n个字符的Huffman编码HC

int ReadFile(char *filename, char *chars);
  // 从文件中逐个字节读取信息

int GetFrequency(char *chars, int *freq, char *charList, int &n);
  // 获取待处理文本字符的频率信息

void WriteCodedString(char *CodedChar, char *chars, HuffmanCode HC, char *CharList, int
    FLength, int CharNum);
  // 将比特流写入文件

void Decode(char *CodedChar, HuffmanCode HC, char *CharList, char *DecodedChar);
  // 根据比特流还原文件
```

3 调试分析

3.1 测试数据

选取纯文本, .mp4 文件, .bmp 文件, .exe 文件进行测试

3.2 问题及解决方法

测试中发现程序无法处理除纯文本文件之外的文件类型，经调试发现数组最大值过小导致读取文件时字符频率的获取不正确。

解决方法：数组长度改为 1000000

4 算法的时空分析

我们可以从以下几个方面分析复杂度:

1. `Select` 函数: 这个函数在 Huffman 树数组 `HT` 中寻找两个最小的未被选中的节点。它通过两次遍历整个数组来实现，因此其时间复杂度为 $O(n)$。
2. `HuffmanCoding` 函数: 这个函数构建 Huffman 树。它首先初始化一个大小为 $2n-1$ 的数组，然后通过 $n-1$ 次迭代来构建树。每次迭代中，它调用 `Select` 函数（$O(n)$），然后执行常数时间的操作。因此，这个函数的总体时间复杂度为 $O(n^2)$。
3. `ReadFile` 函数: 这个函数读取文件内容。它的时间复杂度取决于文件的大小，假设文件大小为 f，则时间复杂度为 $O(f)$。

4. `GetFrequency` 函数: 这个函数计算每个字符的频率。它的时间复杂度为 $O(n^2)$，其中 n 是文件内容的长度。

5. `WriteCodedString` 和 `Decode` 函数: 这两个函数的时间复杂度取决于输入字符串的长度和字符集的大小。在最坏情况下，它们的时间复杂度可以达到 $O(nm)$，其中 n 是输入字符串的长度，m 是字符集的大小。

这个程序的主要瓶颈在于 `HuffmanCoding` 函数和 `GetFrequency` 函数，它们都有 $O(n^2)$ 的时间复杂度。这意味着对于大型输入，这个程序的性能可能会受到显著影响。

5 测试结果及分析

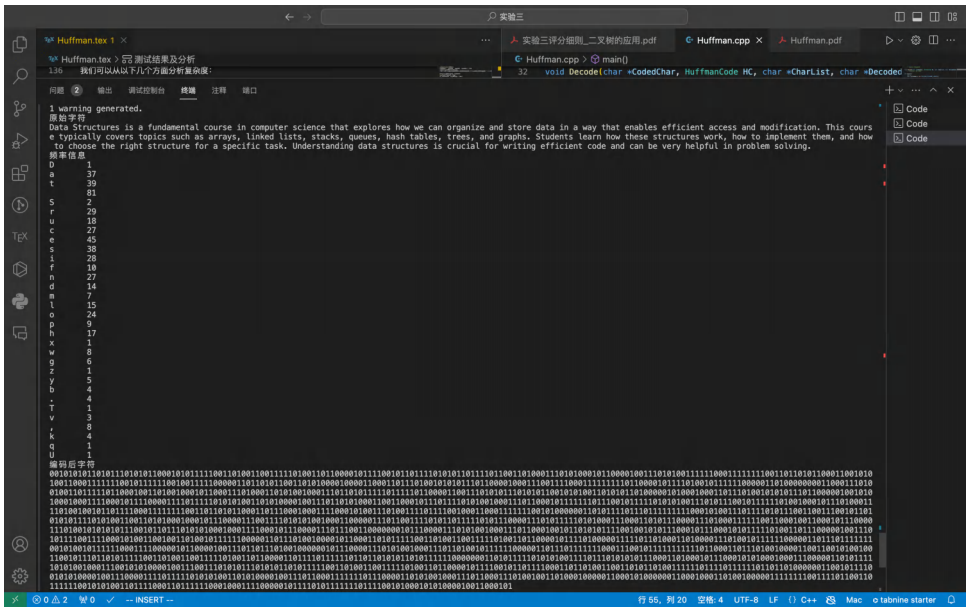
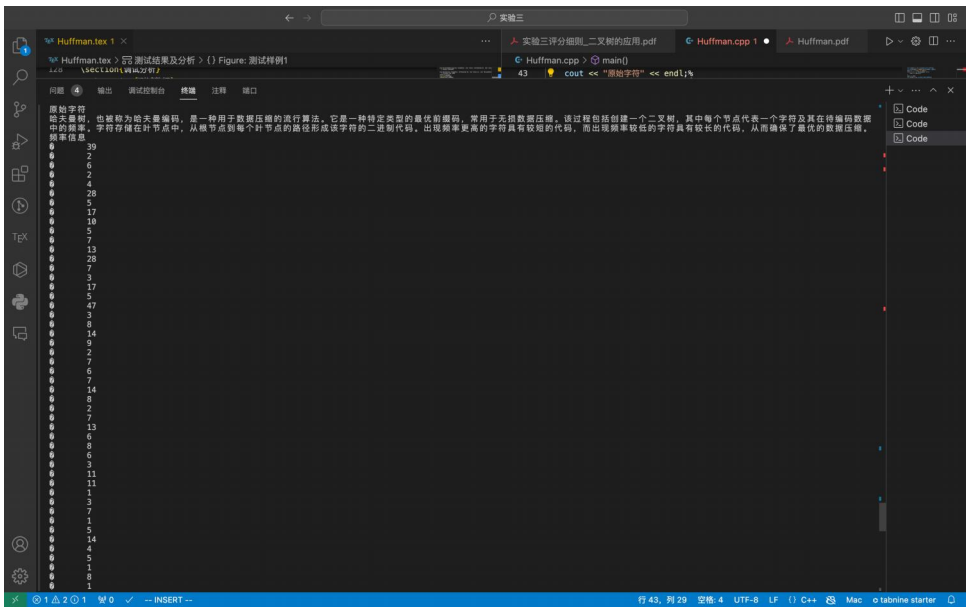


图 1: 测试样例 1



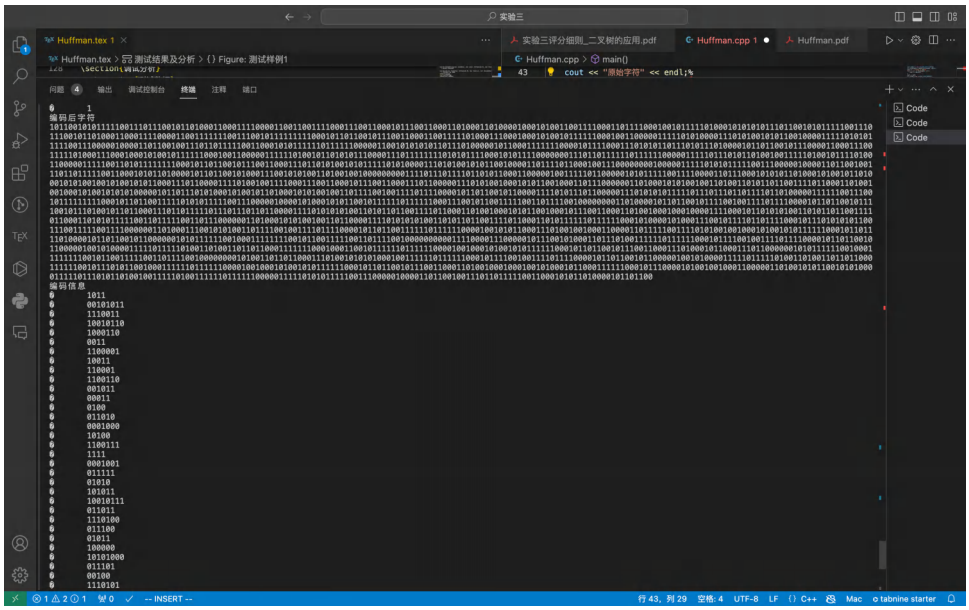


图 2: 测试样例 2

6 实验体会和收获

熟练掌握了二叉树的相关结构、性质，以及 Huffman 算法的原理。

实验四 图

崔士强 PB22151743

1 问题描述

1. 输入一个无向图，输出图的深度优先搜索遍历顺序与广度优先搜索遍历顺序。
2. 输入一个无向铁通讯网图，用 Prim 和 Kruskal 算法计算最小生成树并输出。
3. 输入一个无向铁路交通图、始发站和终点站，用 Dijkstra 算法计算从始发站到终点站的最短路径。

2 算法描述

在三个程序中均使用一个类`Graph`进行操作，其定义分别如下：

2.0.1 图的遍历

```
class Graph
{
    public:
        int VertexNum;
        int EdgeNum;
        int **AdjMatrix; // adjacency matrix
        int *visited; // used for mark visited vertices
        Status InitVisitedArray();
        Status InitAdjMatrix();
        Status InitGraph();
        Status DFS(int start, Stack &S); // recursively travers the graph using a stack
        void DFSCheck(int obj, Stack &S);
        void BFSCheck(int obj, Queue &Q);
        Status BFS(int start, Queue &Q); // recursively travers the graph using a queue
        bool NoAdjVex(int vertex);
        int Search(int vertex);
};
```

2.0.2 最小代价生成树

```
class Graph
{
```

```
    public:
        int VertexNum;
        int EdgeNum;
        int **AdjMatrix;
        int *visited;
        int **ConnecMatrix;
        Status InitVisitedArray();
        Status InitAdjMatrix();
        Status InitGraph();
        int Prim();
        int FindMinimum();
        int Kruskal();
        void FindLowestCost(int &vertex1, int &vertex2);
        void Connect(int vertex1, int vertex2);
        bool AllConnected();
        bool AllVisited();
        int Search(int vertex);
};
```

2.0.3 最短路径

```
class Graph
{
    public:
        int VertexNum;
        int EdgeNum;
        int **AdjMatrix;
        int *visited;
        int *distance;
        Status InitVisitedArray();
        Status InitAdjMatrix();
        Status InitGraph();
        int ShortestPath(int start, int end);
        Status InitDistance(int start);
        int NearestVertex();
        Status Relax(int vertex);
        bool AllVisited();
};
```

3 调试分析

3.1 测试数据

3.1.1 图的遍历

第一行是两个数 n，$m(1 < n < 30\ 1 < m < 300)$，分别表示顶点数量和边的数量接下来的 m 行每行输入两个数 a、b；表示顶点 a 与顶点 b 之间有边相连，顶点编号从 1 到 n 最后输入一个数 s 表示遍历的起始顶点编号

3.1.2 最小代价生成树

第一行是两个数 n，$m(1 < n < 10000\ 1 < m < 100000)$，分别表示顶点数量和边的数量。接下来的 m 行每行输入三个数 a、b、w；表示顶点 a 与顶点 b 之间有代价为 w 的边相连，顶点编号从 1 到 n。

3.1.3 最短路径

第一行是两个数 n，$m(1 < n < 100000\ 1 < m < 1000000)$，分别表示顶点数量和边的数量接下来的 m 行每行输入三个数 a、b、w；表示顶点 a 与顶点 b 之间有长度为 w 的边相连，顶点编号从 1 到 n 最后输入两个数 s，t 表示遍历的起始顶点编号和终点编号

3.1.4 问题及解决方法

1. 在实验过程中，Kruskal 算法无法给出正确答案，经 debug 发现，原因在于判断两节点是否处于同一连通分量的算法出现错误。程序中利用一个二维数组进行标记，每次连接一条边后应当更新矩阵。不仅需要更新被连接的两个点，还需要更新这两个点所在连通分量上所有点.

2. `DFS`和`BFS`在本程序的实现中具有较高的时间复杂度，主要由于搜索邻接顶点的方式导致每次递归或循环调用都有较高的时间成本。优化这一点可以通过使用邻接表代替邻接矩阵来实现，这样可以将搜索邻接顶点的操作从 $O(N_{vertex})$ 降低到 $O(N_{edge})$，从而使`DFS`和`BFS`的时间复杂度接近于其理论最低复杂度 $O(N_{vertex} + N_{edge})$。

4 算法的时空分析

4.1 图的遍历

1. 初始化邻接矩阵（`InitAdjMatrix`）：这个函数创建一个二维数组来存储图的邻接矩阵。由于它包含两层循环，每个循环遍历 N_{vertex} 次，因此其时间复杂度为 $O(N_{vertex}^2)$。

2. 初始化访问数组（`InitVisitedArray`）：这个函数初始化一个标记顶点是否被访问过的数组。这是一个一层循环，时间复杂度为 $O(N_{vertex})$。

3. 初始化图（`InitGraph`）：这个函数读取顶点和边的数量，初始化邻接矩阵和访问数组。它的时间复杂度由初始化邻接矩阵决定，为 $O(N_{vertex}^2)$。

4. 深度优先搜索（`DFS` 和 `DFSCheck`）：`DFS` 的时间复杂度通常为 $O(N_{vertex} + N_{edge})$，因为每个顶点和边都会被访问一次。但由于本程序的实现在搜索邻接顶点时对所有顶点进行了循环，这导致每次递归调用都有 $O(N_{vertex})$ 的时间复杂度，因此总的时间复杂度可能接近于 $O(N_{vertex}^2)$。

5. 广度优先搜索（`BFS` 和 `BFSCheck`）：`BFS` 的时间复杂度与 `DFS` 相似，通常为 $O(N_{vertex}+N_{edge})$。然而，同样由于邻接顶点搜索的实现方式，总的时间复杂度可能也接近于 $O(N_{vertex}^2)$。

6. 搜索未访问的邻接顶点（`Search`）：这个函数对一个顶点的所有邻接顶点进行搜索，时间复杂度为 $O(N_{vertex})$。

4.2 最小代价生成树

1. 初始化邻接矩阵（`InitAdjMatrix`）：这个函数创建并初始化一个二维数组作为邻接矩阵，其时间复杂度为 $O(N_{vertex}^2)$。

2. 初始化图（`InitGraph`）：这个函数读取顶点和边的数据，初始化邻接矩阵和访问数组。整体时间复杂度由邻接矩阵的初始化决定，为 $O(N_{vertex}^2)$。

3. Prim 算法：Prim 算法的时间复杂度取决于如何找到每个顶点的最小权重边。在此实现中，对于每个未访问的顶点，都遍历所有边来找到最小权重边，因此时间复杂度为 $O(N_{vertex}^2)$。

4. Kruskal 算法：Kruskal 算法的时间复杂度取决于边的排序和连接组件的检查。在此实现中，每次都从未连接的边中找到最小权重边，这需要遍历所有边。在最坏情况下，这会导致 $O(N_{vertex}^2)$ 的时间复杂度。

5. `AllConnected` 和 `AllVisited` 方法：这两个方法都有 $O(N_{vertex})$ 的时间复杂度。

6. 连接组件（`Connect`）：在 Kruskal 算法中，每次添加边时都需要更新连接组件。在最坏情况下，这可能需要 $O(N_{vertex}^2)$ 的时间。

Prim 和 Kruskal 算法的时间复杂度在这个实现中都是 $O(N_{vertex}^2)$。这对于较小的图是可行的，但对于大型图可能不够高效。通常，Prim 算法可以通过使用优先队列（如最小堆）来降低时间复杂度到 $O((N_{vertex} + N_{edge}) \log N_{vertex})$，而 Kruskal 算法可以通过对边进行排序（如快速排序或归并排序）以及使用并查集来降低时间复杂度至 $O(N_{edge} \log N_{edge})$

4.3 最短路径

1. 初始化邻接矩阵（`InitAdjMatrix`）：这个函数创建并初始化一个二维数组作为邻接矩阵，其时间复杂度为 $O(N_{vertex}^2)$，其中 N_{vertex} 是顶点的数量。

2. 初始化图（`InitGraph`）：这个函数读取顶点和边的数据，初始化邻接矩阵和访问数组。整体时间复杂度由邻接矩阵的初始化决定，为 $O(N_{vertex}^2)$。

3. 计算最短路径（`hortestPath`）：这个函数使用迪杰斯特拉算法计算从起始点到终点的最短路径。它在每次迭代中找到未访问顶点中距离最短的顶点，然后更新所有邻接顶点的距离。算法的总体时间复杂度为 $O(N_{vertex}^2)$，因为它涉及到对每个顶点的搜索和松弛（Relax）操作。

4. 初始化距离数组（`InitDistance`）：这个函数初始化距离数组，时间复杂度为 $O(N_{vertex})$。
5. 寻找最近顶点（`NearestVertex`）：这个函数遍历所有顶点以找到未访问的距离最短的顶点，时间复杂度为 $O(N_{vertex})$。
6. 更新操作（`Relax`）：这个函数更新与给定顶点相邻的所有未访问顶点的距离，时间复杂度为 $O(N_{vertex})$。
7. 检查所有顶点是否访问过（`AllVisited`）：这个函数检查是否所有顶点都已访问，时间复杂度为 $O(N_{vertex})$。

5 测试结果及分析

5.1 图的遍历

输入：

```
5 6
1 5
1 4
3 1
2 1
2 5
5 3
1
```

输出：

```
1 2 5 3 4
1 2 3 4 5
```

5.2 最小代价生成树

输入：

```
4 5
1 2 2
1 3 2
1 4 3
2 3 4
3 4 3
```

输出：

```
7
```

5.3 最短路径

输入:

```
4 5
1 2 10
1 3 20
2 3 15
2 4 30
3 4 20
1 4
```

输出:

```
40
```

6 实验体会和收获

通过本次实验，掌握了图的存储结构，熟悉了遍历、求最小代价生成树、求最短路径的算法

实验五 哈希表

崔士强 PB22151743

1 问题描述

本实验根据文本文件建立哈希表。过程如下：

1. 输入关键字序列
2. 用除留余数法构建哈希函数，用线性探测法 (线性探测再散列) 解决冲突，构建哈希表 `HT1`
3. 用除留余数法构建哈希函数，用拉链法 (链地址法) 解决冲突，构建哈希表 `HT2`
4. 分别对 `HT1` 和 `HT2` 计算在等概率情况下查找成功和查找失败的 ASL；
5. 分别在 `HT1` 和 `HT2` 中查找给定的关键字，给出比较次数

2 算法描述

本程序定义了两个类来分别进行链地址法和线性探测法操作：

```
 class HashTable_Chaining{
  public:
      HT_data data;
      LNode *links;     // list of links storing the keys
      int *func;        // value of hash fuction of keys
      LinkList *Lists; // list of linklists in the hash table
      int *SearchLength_Succeeded;
      double ASL_Succeeded;
      int *SearchLength_Failed;
      double ASL_Failed;
      void BuildTable(HT_data FileData);
      int Hash_Chaining(int key);
      void GetSearchLength();
      int Search(int key);
      void PrintTable();
};

class HashTable_LinearProbe{
   public:
      HT_data data;
      int *table;
```

```
        int *SearchLength_Succeeded;
        double ASL_Succeeded;
        int *SearchLength_Failed;
        double ASL_Failed;
        void BuildTable(HT_data FileData); // create a table with the given key
        int Hash_LinearProbe(int key); // initialize HashTable
        void GetSearchLength();
        int Search(int key);
        void PrintTable();
};
```

3 调试分析

3.1 测试数据

测试时使用不同除留余数，不同规模的数据进行测试。测试文件中包括：关键字个数 n，关键字 key（这里我们认为关键字 key 就是哈希表中元素对应的哈希函数值），和除留余数法中的 p.

3.2 问题及解决方法

测试过程中发现对于过长的数据，程序给出的结果不正确。经过检查发现原因是读取文件时用于接收数据的数组过小解决方法：

1. 扩大数组
2. 采用动态内存管理

本程序采取第一种方法.

4 算法的时空分析

4.1 读取文件

这个函数的时间复杂度为 $O(n)$，其中 n 是文件中的数据量。

4.2 构建哈希表

1. 链地址法（`HashTable_Chaining::BuildTable`）：构建链地址法哈希表的时间复杂度主要由插入操作决定，总体为 $O(n)$，其中 n 是要插入的元素数量。
2. 线性探测法（`HashTable_LinearProbe::BuildTable`）：构建线性探测法哈希表的时间复杂度也是 $O(n)$。但由于线性探测法可能会遇到多次探测的情况，其性能在高负载时可能略低于链地址法。

4.3 计算查找长度

这个过程的时间复杂度为 $O(n)$，其中 n 是哈希表中的元素数量。这是因为需要遍历哈希表中的每个元素以计算平均成功和失败的查找长度。

4.4 打印哈希表

这个过程的时间复杂度为 $O(m)$，其中 m 是哈希表的大小。这包括打印哈希表中的每个元素和相关的查找长度。

5 测试结果及分析

5.1 测试样例展示

```
----------Beginning of hush1.txt----------
10
1 4 72 107 79 48 118 87 56 126
13
-------------End of hush1.txt-------------
----------Beginning of hush2.txt----------
35
387 390 144 273 149 280 281 413 157 32 35 168 42 44 177 53 438 441 57 448 193 321 451
    329 457 459 76 330 343 226 109 495 369 377 509
53
-------------End of hush2.txt-------------
----------Beginning of hush3.txt----------
47
646 647 139 142 145 657 149 789 535 153 155 157 158 413 670 677 678 550 554 683 428 45
    306 309 184 444 572 64 320 709 330 586 76 335 597 728 603 353 101 743 104 621 368
    626 757 378 637
79
-------------End of hush3.txt-------------
----------Beginning of hush4.txt----------
20
1282 518 1033 1558 672 931 168 1327 687 570 837 843 90 738 1123 747 621 1518 1267 886
191
-------------End of hush4.txt-------------
----------Beginning of hush5.txt----------
160
3 1541 1544 15 528 1044 1558 1058 1571 546 1061 548 42 555 1580 1066 1583 57 1593 1599
    579 584 1615 1103 600 1114 611 1125 626 629 631 121 1657 633 127 641 643 644 135
    1673 650 1676 144 1684 155 1180 1183 1184 1697 1185 163 165 167 1705 1706 1707 176
    691 1716 181 184 1208 698 1227 204 1234 213 734 744 1771 236 1777 1269 1782 247 1275
    252 255 768 1282 1286 1289 778 274 1811 279 1826 1827 1315 1318 1831 297 310 1339
    1851 1341 1860 327 843 337 1361 339 340 1875 1366 1879 858 1888 1378 358 1382 1384
    367 1393 1906 369 887 888 894 1923 1411 899 1930 396 1421 912 913 914 1942 920 1947
    1439 1444 422 1452 941 1966 943 1973 438 440 443 964 1992 1483 973 974 465 1496 2013
    2014 991 2020 1509 1001 1005 1517 1521 498 503
```

```
409
-------------End of hush5.txt-------------
```

5.2 测试结果

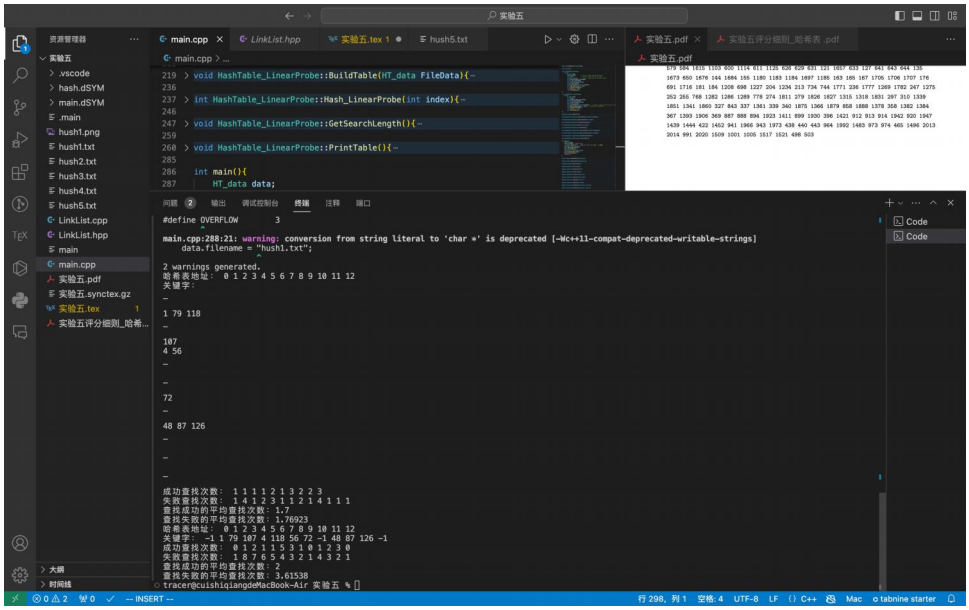


图 1: hush1.txt

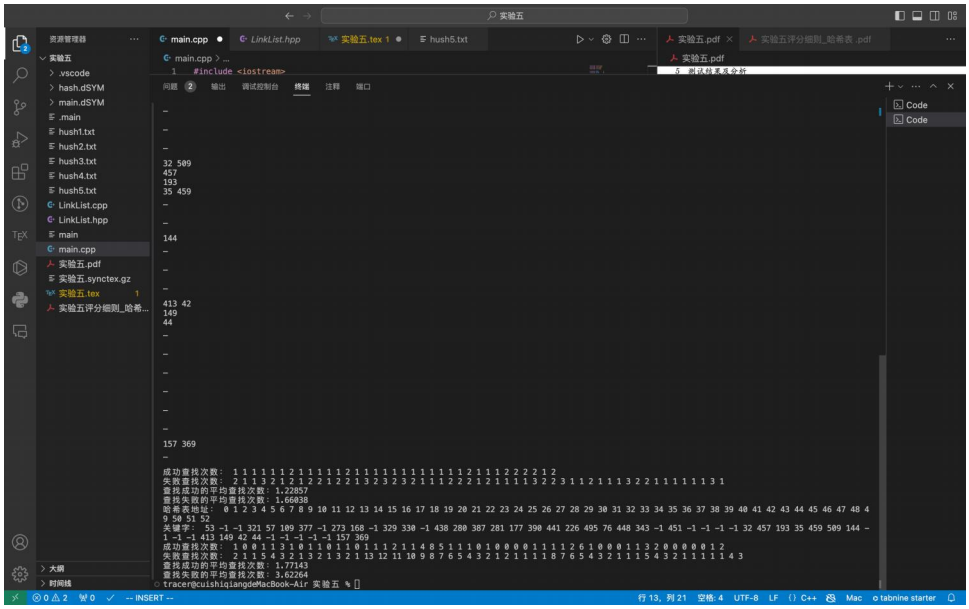


图 2: hush2.txt

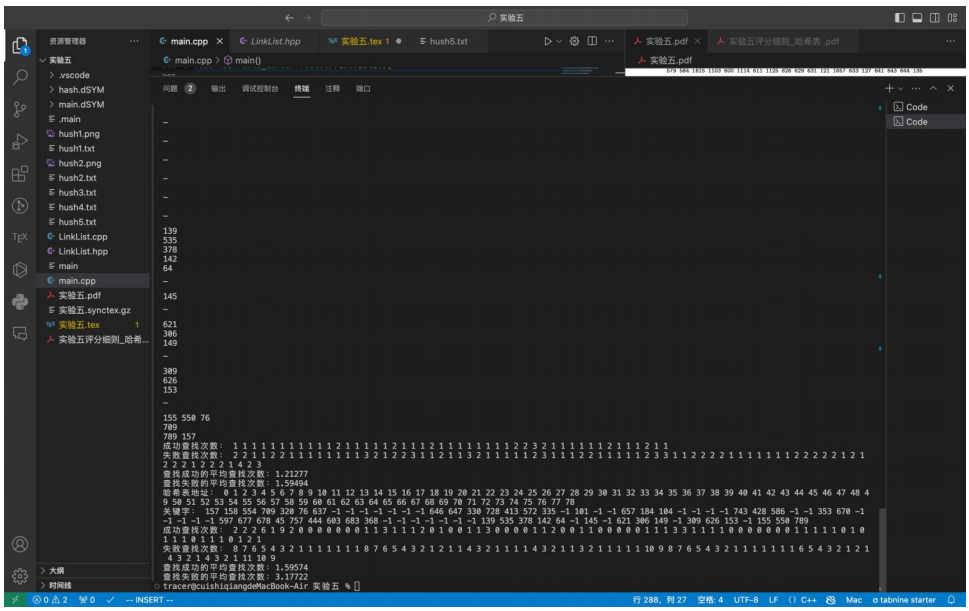


图 3: hush3.txt

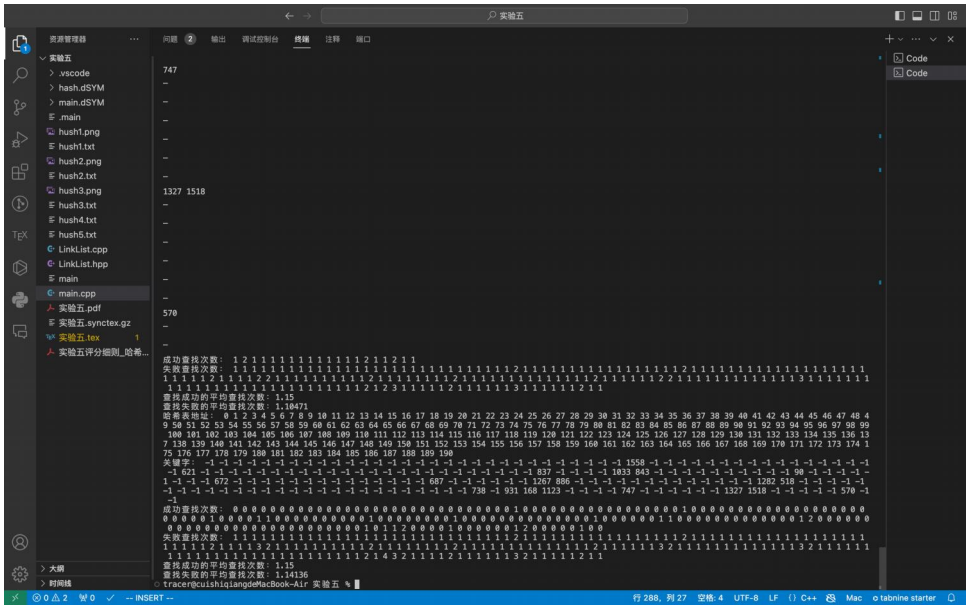


图 4: hush4.txt

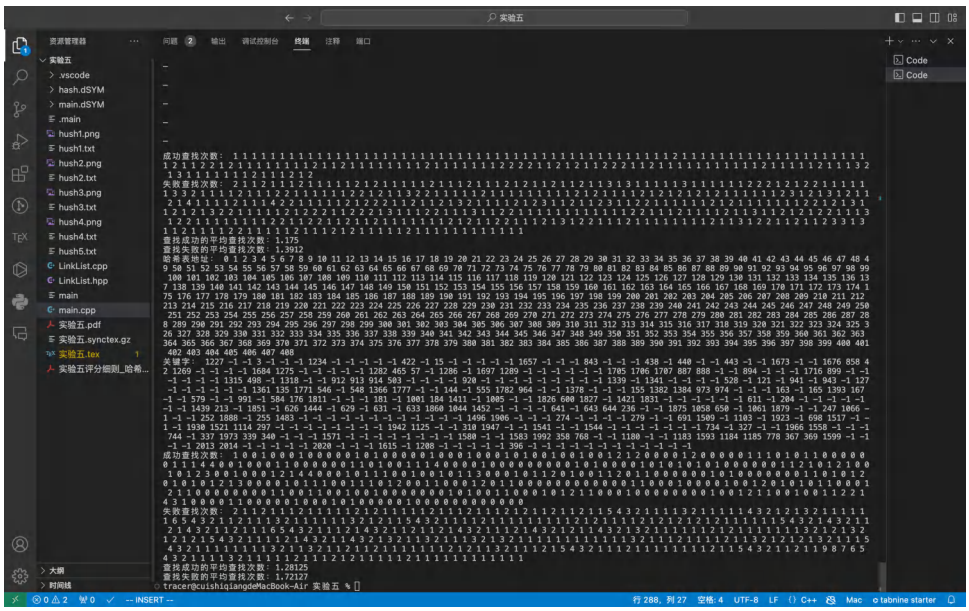


图 5: hush5.txt

结果均正确.

6 实验体会和收获

通过实现两种构建哈希表的方法，理解了哈希表的原理、处理冲突的方式，掌握了存储以及快速查找数据的方法。