数据结构与算法设计

实践总结报告





**专 业**  数字媒体技术

**班 级**

**学 号**

**姓 名**

**日 期**  2022.7.1

东北大学软件学院

问题一 稀疏多项式计算

1. 需求分析

本问题主要实现一元稀疏多项式的运算，实现对于多项式的创建、销毁以及两个多项式间进行相加、相减、相乘的操作并输出运算结果。

多项式的基本设计：一个稀疏多项式是非零元的个数较少的多项式，一般由非零元的个数n和n个非零元素（ai，ei）系数和指数项构成。为了节约存储空间、便于进行运算，采用有序的单链表存储多项式中的非零系数项。

1. 总体设计

**设计思想：**为了实现前文提到的全部功能，我选择将多项式类作为一个.h头文件用include的方式导入.cpp执行文件中。其所有的函数声明和变量定义都在Term.h文件中，具体实现在Term.cpp文件中。

**基本数据结构：**该头文件Term.h中包括：存储多项式结点的Term类，以及存储多项式并进行计算的Polynomial类。Term类及Polynomial类中均有Term\*类型变量，分别用于指向下一个多项式节点及头节点，每个非零多项式结点采用有序的单链表实现存储。

**两个类的定义及函数含义：**

1. **Term类：用于存储多项式节点**

【public类型】

变量声明：

1. float coef：存储单个多项式节点的系数；
2. float pow：存储单个多项式节点的指数；
3. Term\* link：用于指向下一个多项式结点的指针。

函数声明：

1. Term(float c, float p)：构造函数，用于初始化多项式节点对象，其中c表示系数、p表示指数。
2. **Polynomial类：用于存储整个多项式并实现两个多项式间的运算功能**

【private类型】

变量声明：

1. Term\* first：用于存储多项式头节点。

函数声明：

1. void add(Term\* pa, Term\* pb, Term\* pc)：用于进行加法运算；
2. void sub(Term\* pa, Term\* pb, Term\* pc)：用于进行减法运算；
3. void mult(Term\* pa, Term\* pb, Term\* pc)：用于进行乘法运算。

【public类型】

函数声明：

1. Polynomial()：构造函数，用于初始化多项式对象；
2. Polynomial(Polynomial& po)：构造函数，用于初始化多项式对象，可以将Polynomial类型变量po中所有非零元复制到当前新的对象中；
3. ~Polynomial()：析构函数，回收用于存储的链表；
4. void makeEmpty()：回收用于存储的链表；
5. friend istream& operator>>(istream& in, Polynomial& po)：实现输入重载，可以按照题目要求进行输入后将对应数据赋值给当前Polynomial类型对象，包括每个多项式节点的系数及指数；
6. friend ostream& operator<<(ostream& out, Polynomial& po)：实现输出重载，可以按照题目要求的格式输出运算结果，包括每个多项式节点的系数及指数；
7. Polynomial& operator+(Polynomial& po)：实现加号运算符的重载，通过调用add()函数实现加法运算；
8. Polynomial& operator-(Polynomial& po)：实现减号运算符的重载，通过调用sub()函数实现减法运算；
9. Polynomial& operator\*(Polynomial& po)：实现乘号运算符的重载，通过调用mult()函数实现乘法运算。
10. 程序设计

下面将给出重要的函数的算法设计过程。

1. **void add(Term\* pa, Term\* pb, Term\* pc);**

此函数用于加法计算，算法流程图如下图所示。



图 1 void add(Term\* pa, Term\* pb, Term\* pc)函数流程图

1. **void sub(Term\* pa, Term\* pb, Term\* pc);**

此函数用于减法运算，主要思想为将2号多项式所有的系数乘以-1，之后用新的多项式与1号多项式调用add()函数进行加法运算。算法流程图如下图所示。



图 2 void sub(Term\* pa, Term\* pb, Term\* pc)函数流程图

1. **void mult(Term\* pa, Term\* pb, Term\* pc);**

此函数用于乘法运算，主要思想为1号多项式的每个多项式节点分别与2号多项式的所有结点进行相乘操作，即系数相乘、指数相加。之后再将每次相乘的结果调用add()函数进行叠加。算法流程图如下图所示。



图 3 void mult(Term\* pa, Term\* pb, Term\* pc)函数流程图

1. 测试及调试
2. **测试结果**
3. 加法运算测试

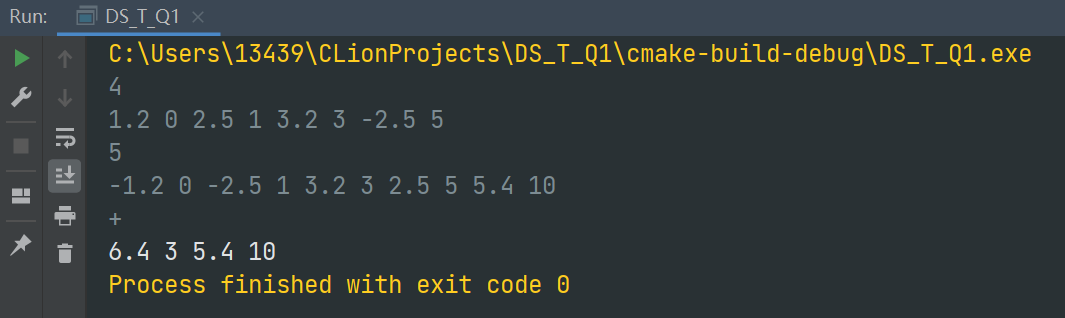


图 4 加法运算测试结果

1. 乘法运算测试

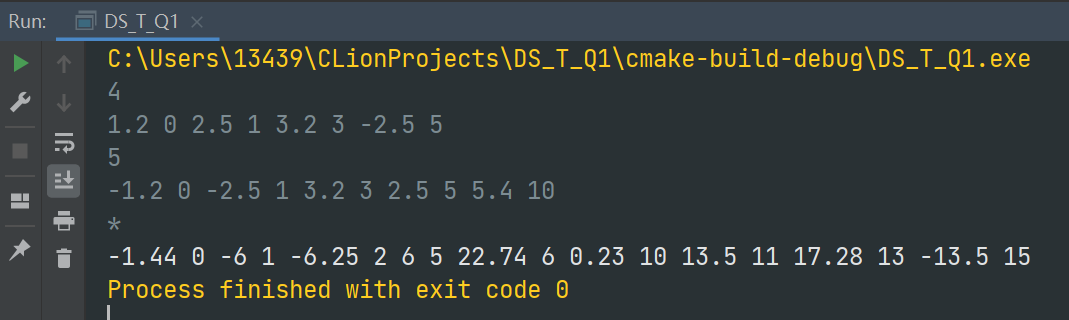


图 5 乘法运算测试结果

1. **调试问题**

在乘法运算函数中，由于调用add()函数中最后一个参数的作用是存储运算结果，而当时前面用过pc变量，忘记对其进行删除，导致结果被重复且混乱地叠加了。

修改：将其赋值为NULL。

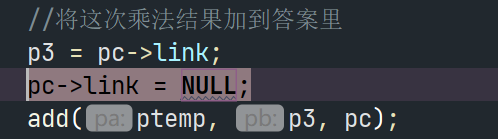


图 6 修改方案

1. 性能分析及结论
2. **性能分析**

对于加法和减法而言，主要运算操作为找到相同指数的多项式节点并对它们进行系数相加、指数不变的操作，共重复n次以完成全部多项式节点，即整个多项式的运算。在这个过程中，开辟了一个Term类型的空间用于结果存储，之后采用链表为其扩展长度，也是n阶的。

因此，加法减法时间复杂度为O(n)，空间复杂度为O(n)

对于乘法而言，其存储结果的空间开辟与加法减法一样，但在运算时涉及到1号多项式的每个节点都需要分别与2号多项式的每个节点进行惩罚操作，即系数相乘、指数不变。之后再将所有的惩罚结果叠加，使空间复杂度上升为平方次。

因此，乘法的时间复杂度为O(n2)，空间复杂度为O(n)。

1. **主要创新点**

采用链式存储结构，链式存储结构的优点为插入、删除十分方便。

1. **系统预留问题**

无。

问题二 哈夫曼编码和译码

1. 需求分析

本问题主要实现了哈夫曼编码和译码功能。在这个问题中，可以建立一棵叶子节点数目n<=26的二叉树，叶子节点的权值分别对应所代表字符在明文中出现的概率。若这棵树的带权路径长度达到最小，则这棵树被称为哈夫曼树，则存储明文时的花费就越少。

本问题具体需求为：编写程序对给定的权值构造一个哈夫曼树，并给定原始字符序列进行编码和译码的测试。

1. 总体设计

**设计思想：**为了实现前文提到的全部功能，我选择将哈夫曼编码类作为一个.h头文件用include的方式导入.cpp执行文件中。其所有的函数声明和变量定义都在HuffmanCode.h文件中，具体实现在HuffmanCode.cpp文件中。

**基本数据结构：**该头文件HuffmanCode.h中包括：代表原节点的结构体Node、用于存储编码结果的编码结构体Hcode以及可以进行创建哈夫曼树、编码、解码等功能的HuffmanCode类。在HuffmanCode类中，采用Node类型的数组ht用于存储原节点的权值、所代表字符、父节点以及左右孩子节点等信息；采用Hcode类型的数组hcode用于存储哈夫曼编码结果。

**两个结构体和一个类的定义及函数含义：**

1. **Node结构体：用于存储原节点。**

变量声明：

1. int weight：用于记录权重值；
2. int parent：用于记录父节点；
3. int left\_child：用于记录左孩子节点；
4. int right\_child：用于记录右孩子节点；
5. char data：用于记录对应的字符。
6. **Hcode结构体：用于存储哈夫曼编码的结果。**

变量声明：

1. char code[50]：用于存储哈夫曼编码；
2. int start：用于记录输出时的起始位置。
3. **HuffmanCode类：用于创建哈夫曼树、进行编码解码等操作。**

【private类型】

变量声明：

1. int n = 0：用于记录需要编码的字符的个数，也是哈夫曼树叶子节点的个数；
2. Node\* ht：用于存储原节点的权值、所代表字符、父节点以及左右孩子节点等信息；
3. Hcode\* hcode：用于存储哈夫曼编码结果。

函数声明：

1. void create\_ht(Node ht[], int n)：根据原节点的权值构造huffman树；
2. void create\_hcode(Node ht[], Hcode hcode[], int n)：将每个字符转换为相应的huffman编码存入编码数组hcode中。

【public类型】

函数声明：

1. HuffmanCode(int a)：构造函数，用于将n的值初始化为a，并创建长度为n的Node类型数组ht以及长度为n的Hcode类型数组hcode，同时接受用户输入将对应节点的权值和所代表字符存入ht中；
2. ~HuffmanCode()：析构函数，删除用于存储的数组ht以及hcode；
3. void create()：调用create\_ht(Node ht[], int n)函数以及create\_hcode(Node ht[], Hcode hcode[], int n)函数，实现构造哈夫曼树并存储字符对应的编码入数组hcode；
4. void encoding(char\* arr)：用于对输入的char类型的arr数组，即要进行翻译的明文序列，进行编码；
5. void decoding(char\* arr)：用于对输入的char类型的arr数组，即要进行译码的报文序列，进行解码。
6. 程序设计

下面将给出重要的函数的算法设计过程。

1. **void create\_ht(Node ht[], int n);**

此函数的先找到最小权重的两个节点，其权重分别记录在min1和min2中，接下来通过比较权重将最小权重的节点记录在left\_node里，第二小权重的节点记录在right\_node里并更新父节点的权重（左右孩子节点权重之和）。重复以上操作建立完整的Huffman树。算法流程图如下图所示。



图 7 void create\_ht(Node ht[], int n)函数流程图

1. **void create\_hcode(Node ht[], Hcode hcode[], int n);**

此函数从叶子节点往根节点进行遍历，对字符进行编码，之后逆序输出则为从根节点至叶节点的顺序。其中左孩子编码值为’0’，右孩子编码值为’1’。将所有的字符都遍历生成对应编码，存入Hcode类型的hc结构体。算法流程图如下图所示。



图 8 void create\_hcode(Node ht[], Hcode hcode[], int n)函数流程图

1. **void encoding(char\* arr);**

此函数直接在通过create\_hcode()函数调用后生成的存储着每个字符及其对应编码的结构体hcode中寻找相应的结果并输出即可。

1. **void decoding(char\* arr);**

此函数从根节点开始遍历，遇到0则进入左子树，遇到1则进入右子树。遍历到叶子节点时输出该叶子节点代表的字符。算法流程图如下图所示。



图 9 void decoding(char\* arr)函数流程图

1. 测试及调试
2. **测试结果**

注：

【输入形式】权值的个数n及n个权值

要进行翻译的明文

要进行译码的报文

【输出形式】明文及其翻译的报文

报文及其翻译的明文

1. 测试一

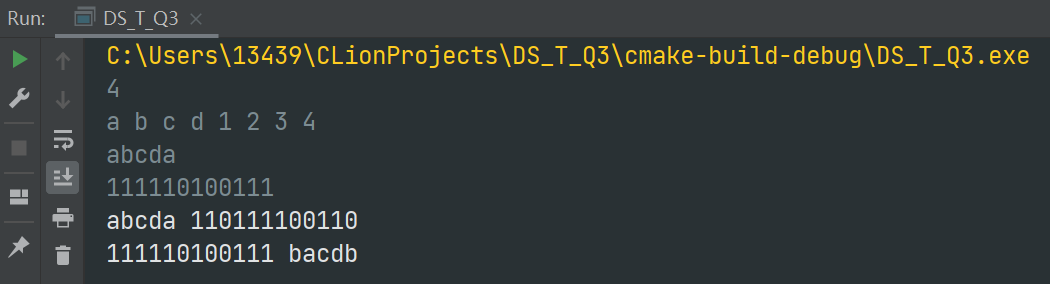


图 10 测试一运行结果

1. 测试二

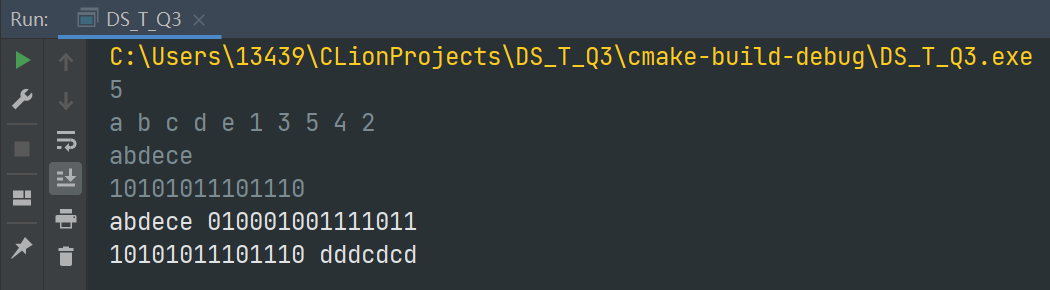


图 11 测试二运行结果

1. **调试问题**

在封装类时，由于有一个变量n用于存储叶子结点的个数，在创建哈夫曼树之前就需要用一个长度为n的Node类型数组ht进行存储。因此最开始写的初始化时的代码为：

**private**:  
 **int** n;  
 Node ht[n];

然而这样其实无法对ht数组进行初始化，会出现问题：

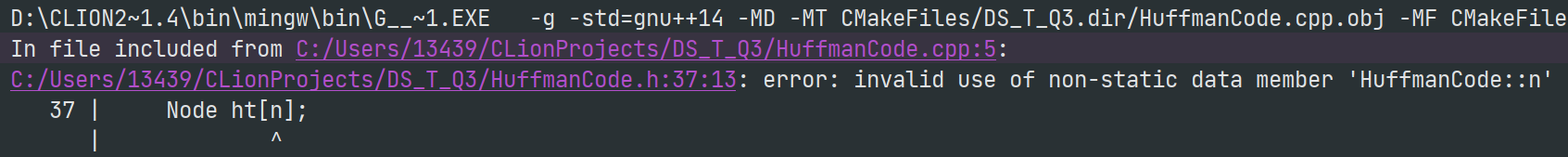


图 12 报错截图

**修改：**考虑在声明数组变量的时候只确定其变量类型，在构造函数中再对其大小进行确定。（数组hcode和数组ht是类似的情况，不再展开）。

修改方式如下：（左边为HuffmanCode.h文件，右边为HuffmanCode.cpp文件）

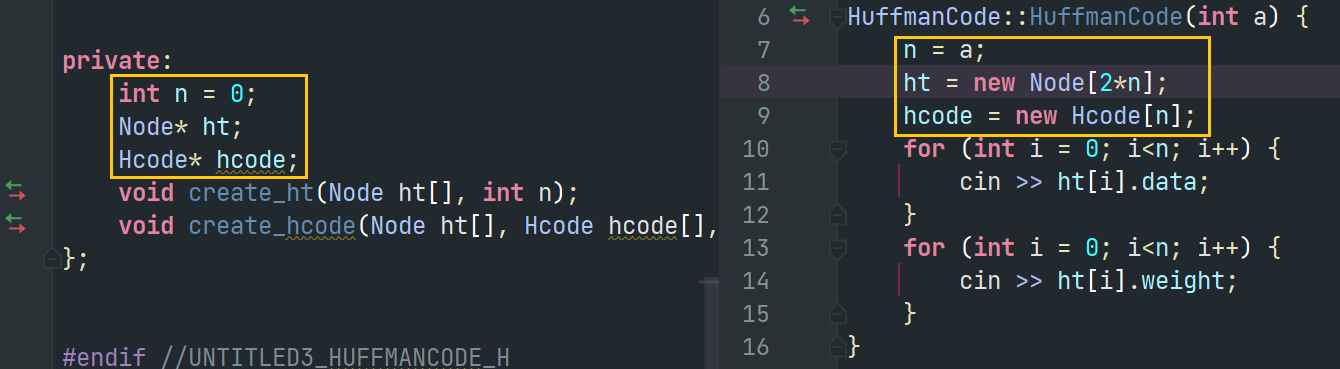


图 13 修改代码截图

1. 性能分析及结论
2. **性能分析**

本问题使用堆排序来构造哈夫曼树用于存储每个字符及其对应的权重，每次迭代需要O(logn)时间来确定最小的权重并插入新的权重。以上的操作需要进行O(n)次。因此总体时间复杂度为O(nlogn)。

由于我为所有符号建立一个双向哈希表，因此编码和解码的空间复杂度都将是常量O(1)。

1. **主要创新点**

采用数组存储哈夫曼树节点的同时，采用堆排序的相应思想，使复杂度由普通哈夫曼建树的O(n2)降低至O(nlogn)，提升了效率。

1. **系统预留问题**

无。

问题三 最短路径

1. 需求分析

本问题主要需要实现在一个边上带权网络中可以求出给定的两个顶点之间的最短路径的功能。需要先编写程序对给定的网络中的边、权值构造一个图，之后求出图中的两个顶点之间的最短路径。

由于给定的图为稀疏图，因此将采用邻接表作为图的存储结构。之后采用Dijkstra算法求解图中两个顶点的最短路径。由题意可知，需要对总票价最小及路径最短两种情况进行求解。

1. 总体设计

**设计思想：**为了实现前文提到的全部功能，我选择将Graph类作为一个.h头文件用include的方式导入.cpp执行文件中。其所有的函数声明和变量定义都在Graph.h文件中，具体实现在Graph.cpp文件中。

**基本数据结构：**该头文件Graph.h中包括：存储图中边（代表路径/权值）的Edge结构体、存储图中节点（代表城市）的City结构体、可以在程序中用于数据存储的栈myStack结构体，以及存储整个图并进行最短路径/最小权值运算的Graph类。City类型中有Edge类型的指针，用于节点间的连接，实现邻接表存储图。

**三个结构体、一个类的定义及函数含义：**

1. **Edge结构体：用于存储图的边（代表路径/权值）**
2. friend class Graph：令Graph类可以直接访问此结构体存储的数据；
3. int dest：用于存储边的终点；
4. float cost：用于存储边的权重值；
5. float dist：用于存储边的路径长度；
6. Edge\* link：用于存储边之间的连接
7. Edge() :link(NULL) {}：初始化时将link赋为NULL。
8. **City结构体：用于存储图的节点（代表城市）**
9. friend class Graph：令Graph类可以直接访问此结构体存储的数据；
10. string name：用于存储图节点的名字（城市名）；
11. bool visited：用于记录当前节点是否被遍历；
12. float mindist：用于记录最短路径；
13. float mindist\_cost：用于记录路径最短时的权重和；
14. float mincost：用于记录最小权重；
15. float mincost\_dist：用于记录权重最小时的路径和；
16. int path：用于记录当前节点出发的路径的终点；
17. Edge\* first：用于存储节点出发的路径；
18. City():first(NULL) {}：初始化时将first赋为NULL。
19. **myStack结构体：用于数据存储的栈**

变量声明：

1. int length：用于记录当前栈内数据的个数；
2. string a[10]：用于存储栈内的数据。

函数声明：

1. void create()：用于初始化length；
2. int size()：用于返回当前栈内数据的数目；
3. int empty()：用于判断当前栈是否为空；
4. void push(string x)：用于从栈顶加入数据；
5. string top()：用于返回栈顶数据；
6. void pop()：用于弹出栈顶数据令length减一。
7. **Graph类：用于存储整个图并进行最小路径/权值的运算**

【private类型】

变量声明：

1. City\* node：用于存储所有的节点（城市）；
2. int n：用于记录图的大小；
3. int maxSize：用于记录图最大的大小。

函数声明：

1. void load()：读取输入文件，将其中信息存储入创建的图的邻接表；
2. void reset()：用于释放数据。

【public类型】

函数声明：

1. Graph(int sz)：构造函数，用于初始化图，通过调用load()函数创建一个有n个节点的图；
2. ~Graph()：析构函数，调用reset()函数，将释放数据；
3. float getCost(int v1, int v2)：用于得到两个节点间的权重；
4. float getDist(int v1, int v2)：用于得到两个节点间的路径长；
5. int getCityPos(string name)：用于得到某个节点对应的序号；
6. void ShortCostPath(int v)：用于找到当前节点到某个节点的最小权重的路径；
7. void ShortDistPath(int v)：用于找到当前节点到某个节点的最短距离的路径；
8. void outputCostResult(int v1, int v2)：用于输出最小权重路径；
9. void outputDistResult(int v1, int v2)：用于输出最短距离路径；
10. 程序设计

下面将给出重要的函数的算法设计过程。

1. **void ShortCostPath(int v);**

此函数实现了Dijkstra算法，大致步骤为：先找到节点1并得到V1出发的邻接点的小权重，将节点1加到S集合（即City类型的数组node）中；之后从S集合之外的点中找到权重最小者，对以其为出发点的邻接点进行松弛操作，若权重被更新，则记录前驱；之后重复上一步，直到所有点被S集合收录。

在更新权重的时候，本函数会同步更新距离，以实现最小权重情况下同时输出对应的距离之和。算法流程图如下图所示。



图 14 void ShortCostPath(int v)函数流程图

1. **void ShortDistPath(int v);**

此函数同ShortCostPath(int v)原理一样，只是将cost（权重）和dist（距离）交换，实现最短距离情况的运算。

1. **void outputCostResult(int v1, int v2)**

此函数用栈数据结构实现找到所有利用ShortCostPath(int v)函数找到的路径上的节点，并按序输出。

1. **void outputDistResult(int v1, int v2)**

此函数用栈数据结构实现找到所有利用ShortDistPath(int v)函数找到的路径上的节点，并按序输出。

1. 测试及调试
2. **测试结果**

本程序的输入为arc.txt文件，其第一列为起始城市名称，第二列为终点城市名称，第三列为两个城市间的权重值，第四列为两个城市的距离。文件前5行展示：



图 15 被读取文件的前五行展示

测试结果展示：

1. 测试一：起点为Rome城市，终点为Warsaw城市。

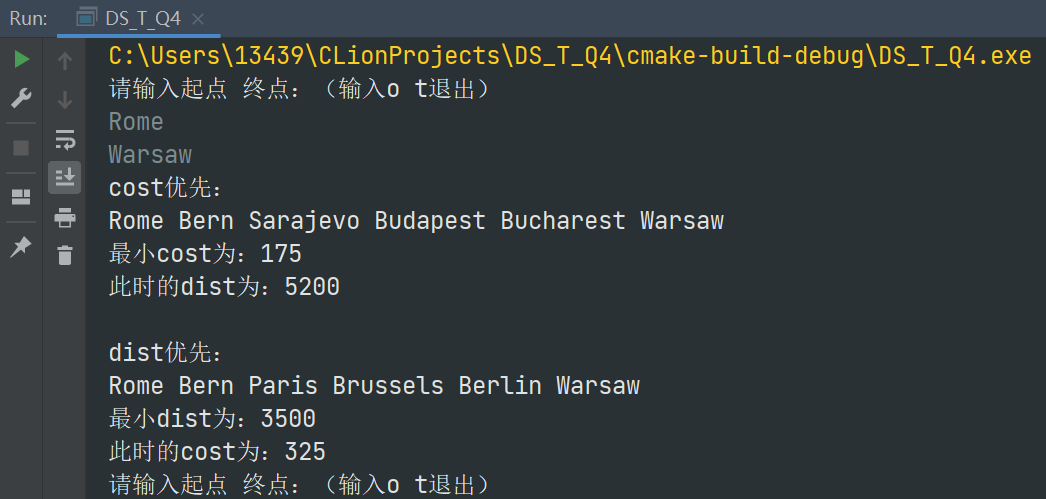


图 16 测试一测试结果

1. 测试二：起点为Rome城市，终点为Bucharest城市。

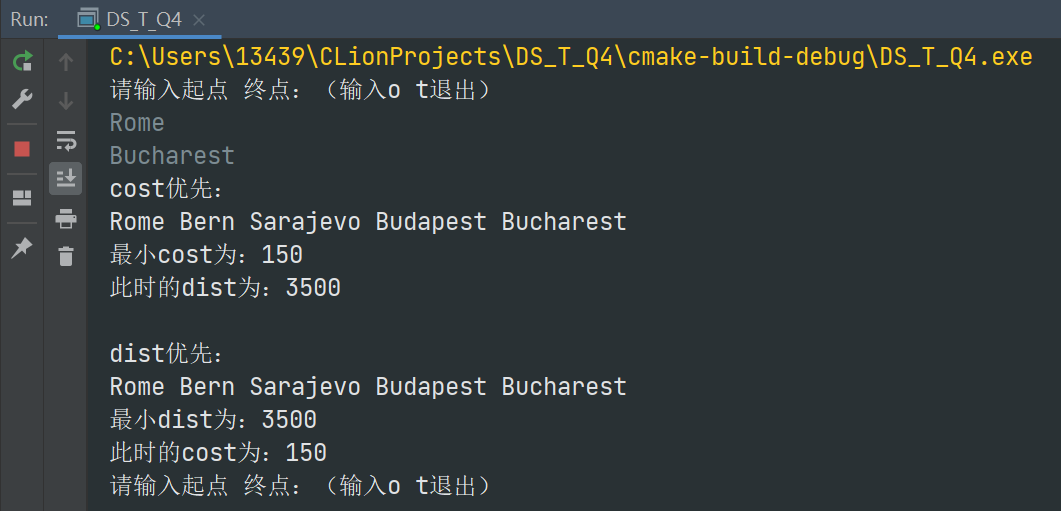


图 17 测试二测试结果

1. **调试问题**

在输出路径的节点时，我选择采用栈数据结构。但在测试时，发现sCost栈和sDist栈被指出“multiple definition”，如下图所示：

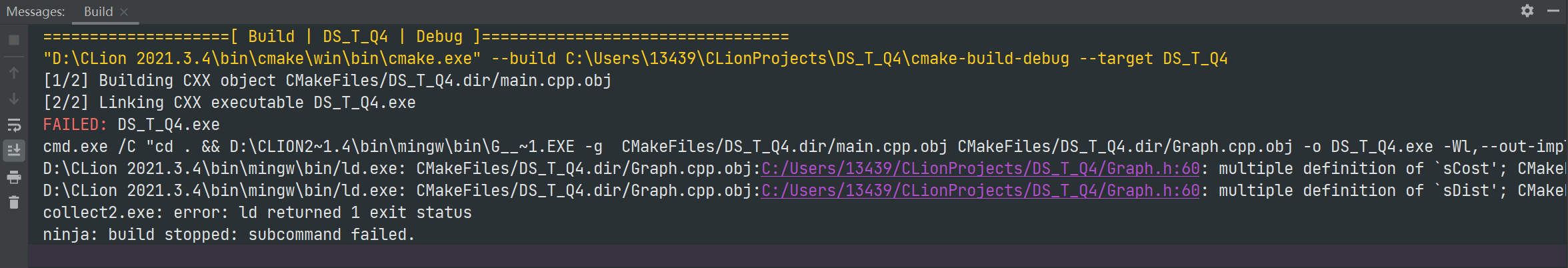


图 18 程序报错截图

我检查了表示栈存储结构的myStack结构体，发现sCost和sDist在Graph.h文件中进行了定义，但在Graph.cpp中使用时没有进行定义。如下图所示：

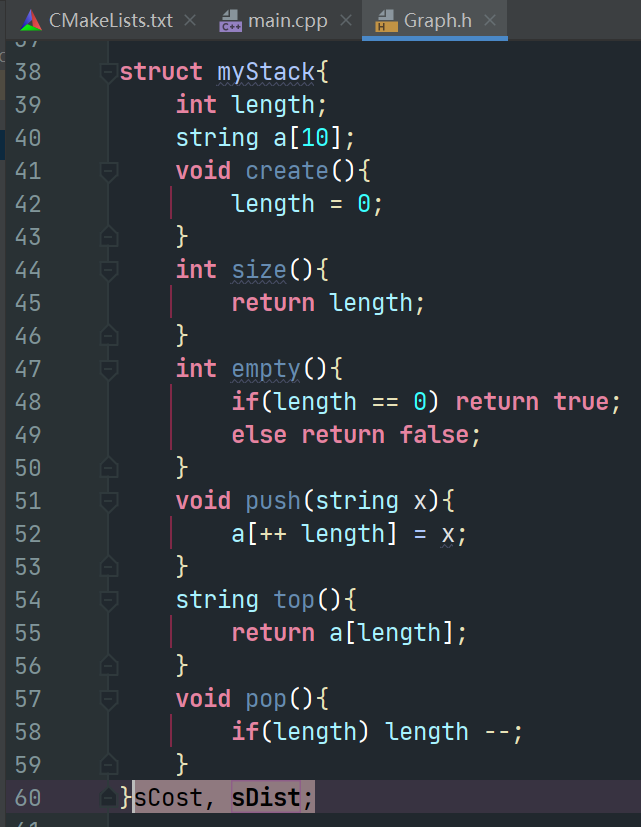


图 19 错误代码展示

**修改：**我将Graph.h文件中的声明（即上图中选中部分）删除，在Graph.cpp文件中需要使用时重新定义myStack类型的对象sCost。sDist同理。改动如下图所示：

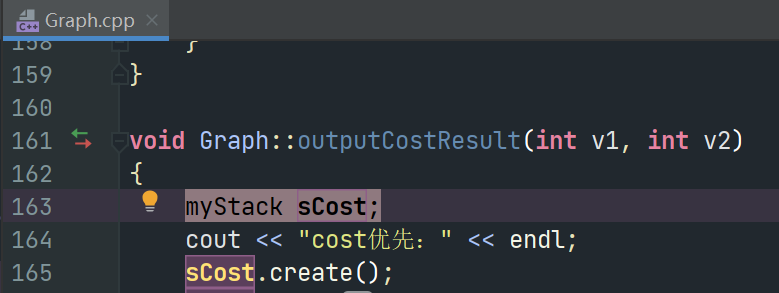


图 20 修改代码展示

1. 性能分析及结论
2. **性能分析**

本问题使用Dijkstra算法来实现最短路径的寻找。其中初始化路径及节点的时间复杂度均为O(n)。在外层循环中，每次需要删除一个顶点，共有n个顶点，因此时间复杂度为O(n)。在内层循环中的搜索最小元素运算复杂度为O(n)，遍历邻居的复杂度为O(v)（其中v为每个节点最大的邻居数）。综上，整个算法的时间复杂度为O(n)+O(n2)+O(vn) ~ O(n2)。

这个程序的实现过程中采用了邻接表存储图，因此其空间复杂度为O(n)。

1. **主要创新点**

采用邻接表存储图，大大节省了空间；

实现了两种情况下最优路径的输出，分别为路径最短以及权重最小。

1. **系统预留问题**

算法可以采用堆优化[1]，使时间复杂度由O(n2)降低至O(n(logn+e))，提升了效率。但此种需要用到优先队列，不采用STL库不太方便实现，尝试过但失败了(qvq)，就没有实现。

问题四 堆排序

1. 需求分析

本问题采用二叉堆数据结构，实现对给定的整数序列进行排序。

1. 总体设计

**设计思想：**为了实现二叉堆排序，我设计了一个HeapSort类，内部的两个主要函数分别实现了以递归方式构建大根堆和堆排序。其所有的函数声明和变量定义都在HeapSort.h文件中，具体实现在HeapSort.cpp文件中。

**基本数据结构：**在HeapSort类中，采用整型变量n存储想进行排序的数字个数，采用整型数组arr存储所有待排序的数字，并在排序过程中对其中所存储的数字进行位置的调整。

**类的定义及函数含义：**

1. **HeapSort类：用于实现堆排序。**

【private类型】

变量声明：

1. int n：用于存储想进行排序的数字的个数；
2. int\* arr：用于存储所有待排序的数字，并在排序过程中对所存储的数字进行位置调整；

函数声明：

1. void adjust(int a, int index)：以递归的方式构建大根堆；
2. void heapSort()：构建大根堆并对其通过调用adjust函数进行调整。

【public类型】

函数声明：

1. HeapSort(int a)：构造函数，初始化长度为n的用于存储数字的数组arr并为其赋值个数为a的随机数字（其中a=n）；
2. ~HeapSort()：析构函数，删除arr数组；
3. void sort()：堆排序函数，通过调用heapSort()函数实现，同时可以记录运行时间。
4. 程序设计

下面将给出重要的函数的算法设计过程。

1. **void adjust(int a, int index);**

此函数采用递归的方式构建大根堆，利用数字在数组中存储所在的位置序号之间的位置关系表示堆内节点之间的关系。算法流程图如下图所示。



图 21 void adjust(int a, int index)函数流程图

1. **void heapSort();**

此函数先从最后一个非叶子节点向上调用adjust(int a, int index)函数构建大根堆，之后将当前最大的放置到数组末尾固定住，对剩余值调用adjust(int a, int index)函数进行大根堆的调整。算法流程图如下图所示。



图 22 void heapSort()函数流程图

1. 测试及调试
2. **测试结果**

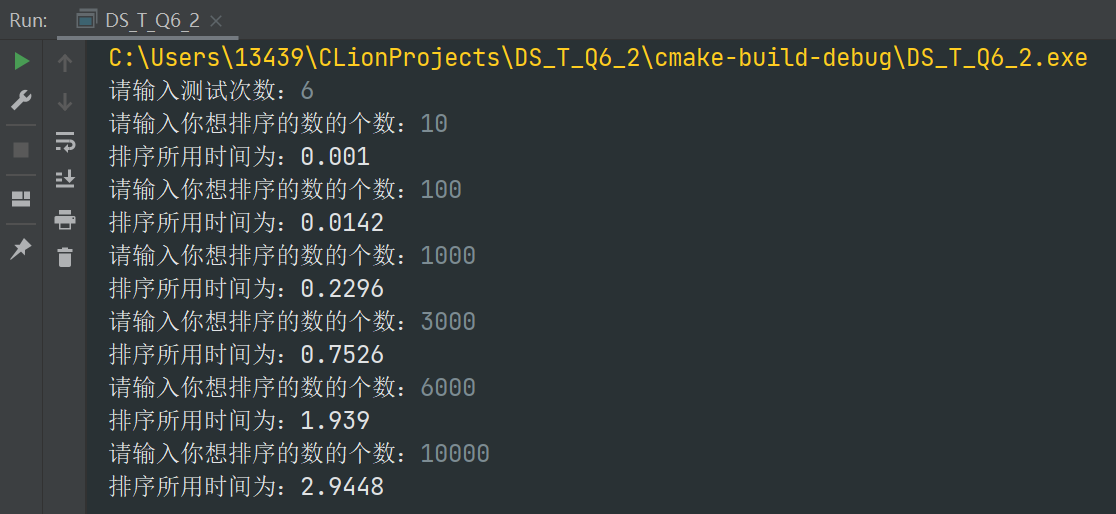


图 23 程序测试截图

由测试结果可以得到堆排序所用时间与排序规模n之间的关系，其趋势如下图所示：

图 24 堆排序所用时间与数据规模关系图

1. **调试问题**

在运行中出现无运行结果但程序中断的现象，如下图所示。

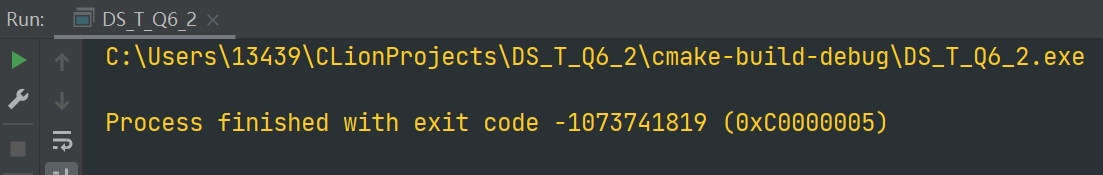


图 25 程序中断截图

之后对程序进行debug，发现整型数组arr在运行过程中只有一个值，而非我需要的n个值。进而导致所有的数字均未被存储，造成程序中断。Debug截图如下所示。

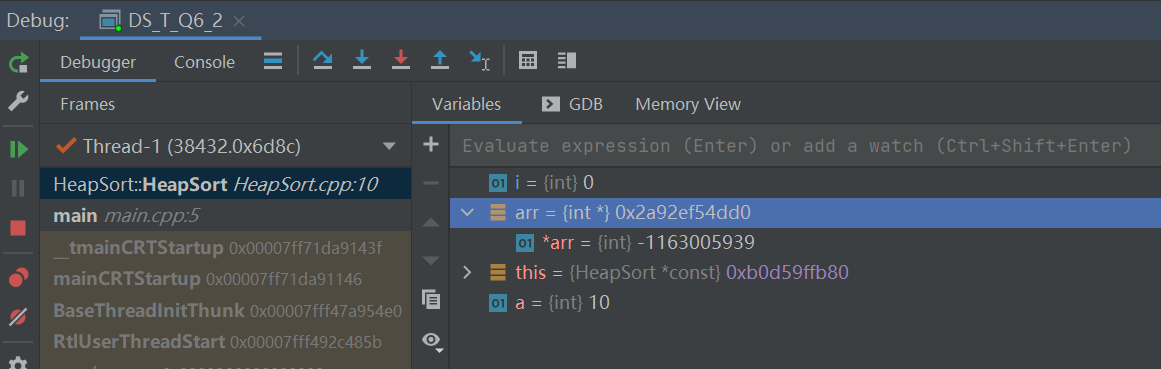


图 26 程序debug截图

我分析问题主要出现在我在构造函数中又新建了一个整型数组arr，与我在private内声明的数组重复，造成以上冲突。猜想错误代码如下所示。

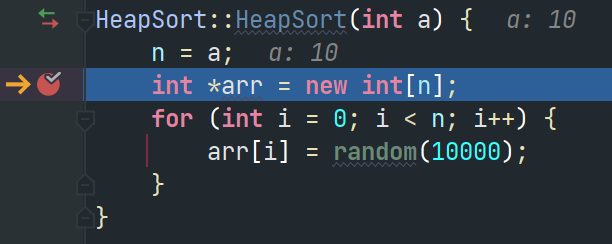


图 27 猜测错误的代码截图

修改：将int\*去掉，等号后面的直接对private中声明的整型数组arr进行赋值即可。修改的代码如下图所示。

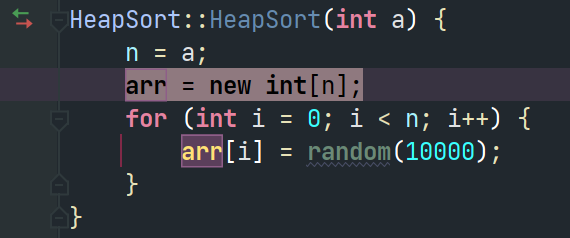


图 28 修改代码截图

经测试，程序可正确运行了。

1. 性能分析及结论
2. **性能分析**

本问题采用了数组存储所有的堆中的节点，通过调用void adjust(int a, int index)函数进行堆化，其时间复杂度为O(logn)。创建堆的过程即调用heapSort()函数，其时间复杂度为O(n)。由于heapSort()函数在调用时会进行n阶次的堆化操作，因此堆排序的整体时间复杂度为O(nlogn)。

同时，开辟了数组空间存储堆内的所有节点，由于这里实现的堆排序为就地排序，因此空间复杂度为O(1)。

1. **主要创新点**

由于堆排序中的二叉树为完整二叉树，因此可以采用数组存储每个节点，并利用每个节点的序号之间的关系来表示堆中左孩子、右孩子及父节点的关系。实现简便且好理解，空间复杂度也为常数阶O(1)。

1. **系统预留问题**

无。

问题五 最小生成树Prim算法

1. 需求分析

此问题需要采用邻接矩阵存储图，并采用Prim算法对给定的图求解最小生成树。

1. 总体设计

**设计思想：**为了实现前文提到的全部功能，我选择将Prim类作为一个.h头文件用include的方式导入.cpp执行文件中。其所有的函数声明和变量定义都在Prim.h文件中，具体实现在Prim.cpp文件中。

**基本数据结构：**该头文件Prim.h中包括：存储整个图并进行最小生成树求解的Prim类。采用二维数组G进行图的邻接矩阵结构存储，n则代表图的节点个数。

**类的定义及函数含义：**

1. **Prim类：用于存储整个图并进行最小生成树求解：**

【private类型】

变量声明：

1. int n = 0：用于记录图的节点个数；
2. int \*\*G：用于存储图的邻接矩阵。

【public类型】

函数声明：

1. Prim(int a)：构造函数，将a赋值给n，用于初始化节点个数为n的图；
2. ~Prim()：析构函数，删除邻接矩阵存储；
3. void run()：运行函数，即用Prim算法求解最小生成树。
4. 程序设计

下面将给出重要的函数的算法设计过程。

1. void run();

此函数采用Prim算法求解最小生成树。主要思想为：从start节点开始，在未被加入到最小生成树的顶点中，找出连通路径权值最小的顶点加入最小生成树，该节点标记为已加入。重复以上操作，最终生成完整的最小生成树。算法流程图如下图所示。



图 29 void run()函数流程图

1. 测试及调试
2. **测试结果**

注：

【输入形式】若干行

第1行：图的结点数量n，本行以回车结束。

第2~n+1行：图的邻接矩阵。矩阵的每行上的数据用空格隔开，每行用回车隔开。

第n+2行：算法开始的结点编号。

1. 测试一

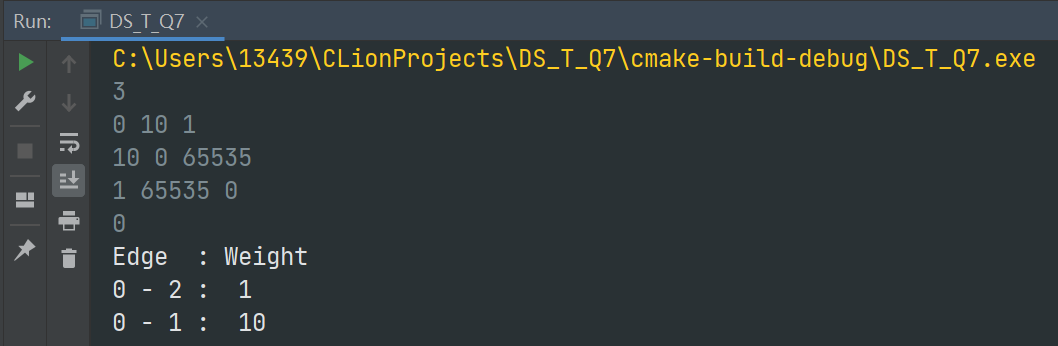


图 30 测试一结果截图

1. 测试二

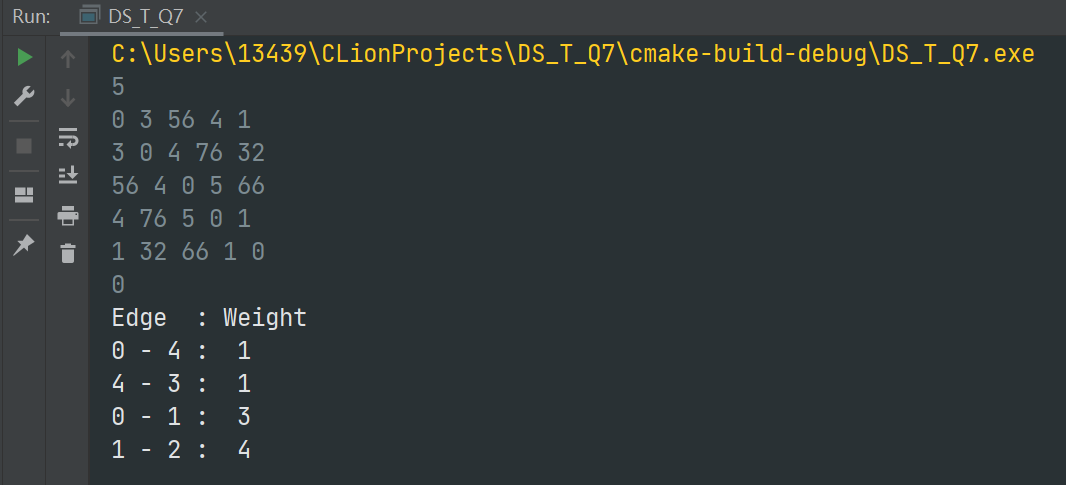


图 31 测试二结果截图

1. **调试问题**

最开始在测试的时候发现有一条边的权重值非常大，猜想是循环的过程中有变量的处理不当。错误输出结果如下：

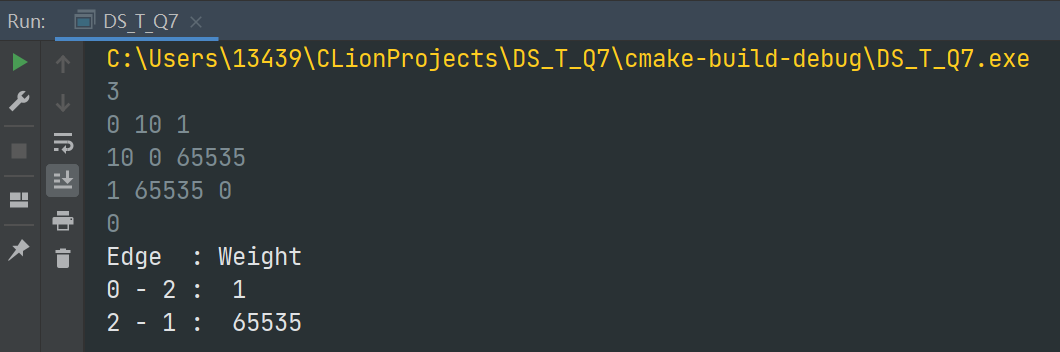


图 32 错误运行截图

经检查，此处的first记录的节点序号在循环过程中会超出节点个数，导致上述问题。

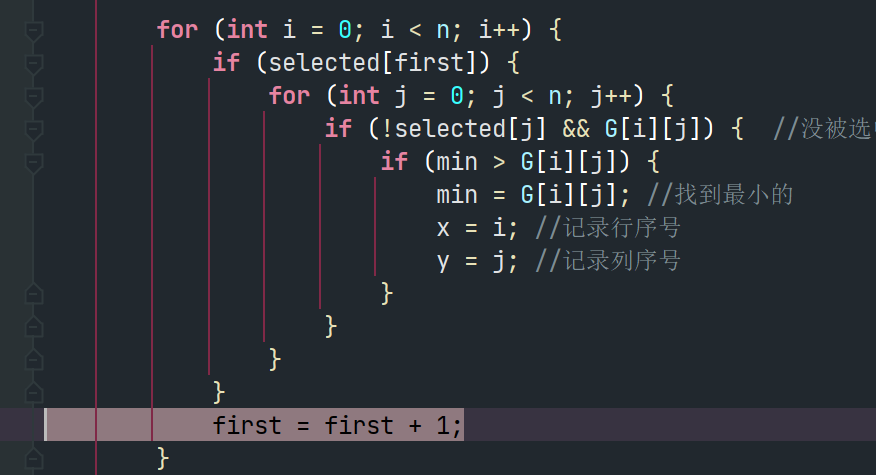


图 33 错误代码截图

**修改：**将其对n取余数，这样就可以继续遍历还未遍历的节点。修改代码如下：

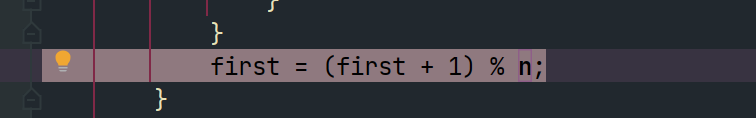


图 34 修改代码截图

经检查，可以正确运行了。

1. 性能分析及结论
2. **性能分析**

本问题采用了邻接矩阵结构来存储图，其有n个顶点。在用Prim算法求解最小生成树时，第一个初始的循环语句频度为n，第二个找n-1条边的循环语句频度为n-1, 其中有两个内循环：其一是求具有最小代价的边，其频度为n, 其二是重新选择具有最小代价的边，其频度为n。因此，Prim算法总的时间复杂度为O(n2)， 与网中的边数无关。因此适用于求边稠密图的最小生成树。

由于采用了邻接矩阵存储图，运用Prim算法的空间复杂度为O(n2)。

1. **主要创新点**

无，主要就实现了题目中所提的用邻接矩阵存储图并对其采用Prim算法求解最小生成树。

1. **系统预留问题**

无法输出所有的最小生成树，我尝试采用了递归算法，但由于时间不足了未成功实现(qvq)。

问题六 词典的实现

1. 需求分析

此问题采用哈希表实现词典的应用，可以对单词进行拼写检查和拼写错误修改建议。主要实现采用哈希表存储字典以及对给定的文章进行检查并找到拼写错误。

1. 总体设计

**设计思想：**为了实现前文提到的全部功能，我选择将wordCheck类作为一个.h头文件用include的方式导入.cpp执行文件中。其所有的函数声明和变量定义都在wordCheck.h文件中，具体实现在wordCheck.cpp文件中。

**基本数据结构：**该头文件wordCheck.h中包括：创建存储所有单词的哈希表并可以实现对文章进行检查并改正错误单词的wordCheck类。采用string类型数组进行哈希表的存储，每个位置的序号对应其位置上字符串（单词）的通过自定义的取值方法生成的取值。

**类的定义及函数含义：**

1. **wordCheck类：实现采用哈希表存储字典以及对给定的文章进行检查并找到拼写错误**

【private类型】

变量声明：

1. int total = 0：用于记录错误单词的个数；
2. string\* h：用于存储构建的哈希表；
3. string charBox[26]：存储了26个字母，方便其他函数调用。

函数声明：

1. int repeatError(string wrongWord)：用于找到因为字母重复引起错误的单词；
2. int swapError(string wrongWord)：用于找到因为字母间交换引起错误的单词；
3. int lossError(string wrongWord)：用于找到因为字母丢失引起错误的单词。

【public类型】

变量声明：

1. const int maxn = 109584：用于记录词典中最大的单词数目。

函数声明：

1. wordCheck()：构造函数，用于初始化实现哈希表的string类型数组h；
2. ~wordCheck()：析构函数，用于删除数组h；
3. int Hash(string x)：对每个单词按照特定的取值规律进行取值；
4. void Insert(string x)：将单词插入到哈希表（数组h）中；
5. bool Find(string x)：用于寻找单词x；
6. void buildHashmap()：用于构建哈希表；
7. void findWrongwords()：通过调用repeatError(string wrongWord)函数、swapError(string wrongWord)函数以及lossError(string wrongWord)函数找到错误的单词并给出改正方案。
8. 程序设计
9. **int Hash(string x);**

此函数主要采用一种取值规则对每个单词进行取值。取值规则为：取值 =（单词第一个字母的ASCII编码 - 65）\*10000 +（单词中间的一个字母的ASCII编码 - 65）\*100 + 单词最后一个字母的ASCII编码 - 65。

1. **void Insert(string x);**

此函数调用Hash(string x)函数得到单词对应的取值，直接将字母添加到字符串类型数组h的取值位。但由于有些单词的首字母、中间字母和末尾字母相同，因此可能出现该位置已经有单词的情况。此时为取值循环+1，直至位置为空的时候添加到位置中。算法流程图如下图所示。



图 35 void Insert(string x)函数流程图

1. **bool Find(string x);**

此函数用于找到单词x。先对单词x按照取值方法进行取值，由于前面的哈希表中已经存储了，因此直接到取值的位置寻找。若为空，则字典中没有这个单词。

1. **int swapError(string wrongWord);**

此函数用于找到由于字母交换引起错误的单词。主要思想为对所有错误的单词的全部相邻两个字母进行交换，看交换后的单词是否在但此表中。若在，则改正成功，进行输出；若不在，则继续尝试其他的错误形式。算法流程图如下图所示。



图 36 int swapError(string wrongWord)函数流程图

1. int repeatError(string wrongWord);

此函数用于找到由于字母重复引起错误的单词。和swapError(string wrongWord)函数类似，也是对单词进行字符串的变换，不再赘述。

1. int lossError(string wrongWord);

此函数用于找到由于字母丢失引起错误的单词。和swapError(string wrongWord)函数类似，也是对单词进行字符串的变换，不再赘述。

1. 测试及调试
2. **测试结果**

注：本程序直接读入文件“dic.txt”以及文件”doc.txt“进行求解。前者为字典，后者为需要纠错的文段。字典中共有109583个单词，如下图所示。



图 37 字典文件

需要纠错的文段如下图所示。

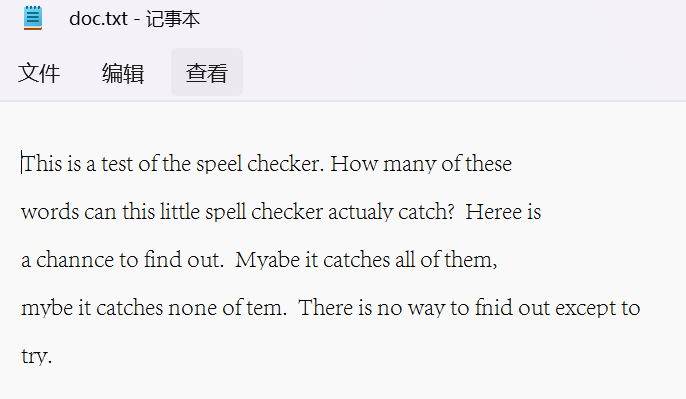


图 38 需要纠错的文段

运行结果：

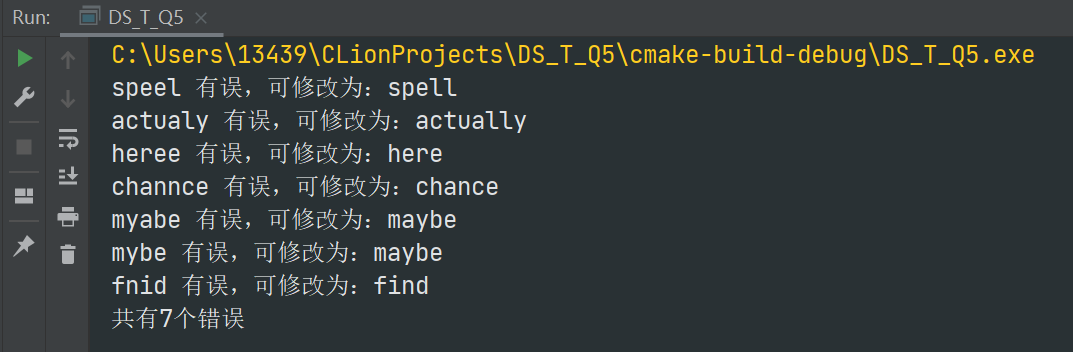


图 39 运行结果截图

1. **调试问题**

一开始在对单词进行取值的时候没有进行加权操作，导致取值过大，出现程序崩溃现象，如下图所示：

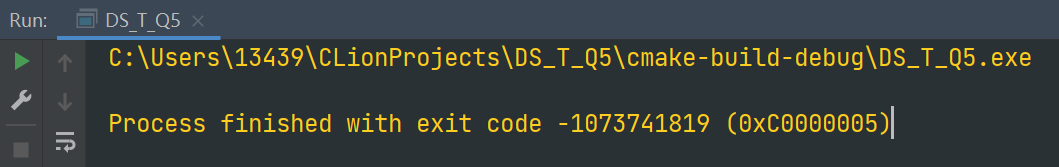


图 40 程序崩溃截图

**修改：**将取值进行加权，修改代码如下：

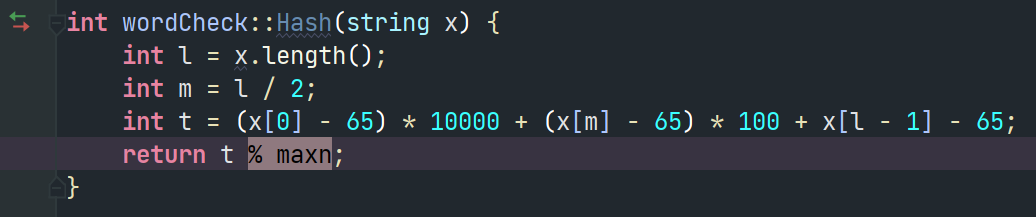


图 41 修改代码截图

经测试，程序可正常运行。

1. 性能分析及结论
2. **性能分析**

本问题采用了哈希表来存储词典哈希表），是根据关键码值而直接进行访问的数据结构。也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。其时间复杂度为O(1)。

哈希表存储字符的空间取决于字符串的字符集大小，最坏情况下每个字符均不相同，需要O(|Σ|)的空间，其中Σ是字符串的字符集[2]。

1. **主要创新点**

将单词错误分为三种，分别是字母交换错误、字母重复错误和字母缺失错误，并分别对三种错误进行查找和改正。

在整体存储上采用哈希表存储，查找单词的复杂度维持在常数阶，十分迅速。

1. **系统预留问题**

无。

参考文献

1. <https://blog.csdn.net/weixin_39763683/article/details/112111509>
2. <https://blog.csdn.net/m0_50363378/article/details/111823053?utm_medium=distribute.pc_relevant.none-task-blog-2~default~baidujs_baidulandingword~default-0-111823053-blog-124148678.pc_relevant_aa2&spm=1001.2101.3001.4242.1&utm_relevant_index=3>

成绩评定表

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 评分项目 | 考核点 | 分数 | **得分** | 备注 |
| 课上表现 | 是否按时上课  是否有缺席 | 20 | **0** |  |
| 报告完整性 | 设计阐述是否完整  测试数据设计是否合理  运行结果是否正确 | 30 | **0** |  |
| 设计实现 | 逻辑是否正确  程序可读性 | 30 | **0** |  |
| 算法难度 | 程序设计是否具有创新性 | 10 | **0** |  |
| 答辩讲解 | 阐述是否清晰  回答是否正确 | 10 | **0** |  |
| 合计 |  | 100 | **0** |  |