

数据结构与算法 课程实验报告

学号：202000130198	姓名：隋春雨	班级：20.4
实验题目：排序算法		
实验学时：2	实验日期：2021-10-21	
实验目的： 掌握稀疏矩阵结构的描述及操作的实现。		
软件开发环境： CLION2020		
1. 实验内容 1、题目描述： 创建稀疏矩阵类（参照课本 MatrixTerm 三元组定义），采用行主顺序把稀疏矩阵非 0 元素映射到一维数组中，实现操作：两个稀疏矩阵相加、两个稀疏矩阵相乘、稀疏矩阵的转置、输出矩阵。 重置矩阵：操作 1，即重置矩阵 P 的尺寸为 n 行 m 列，且随后按行优先顺序输入矩阵 P 的各个元素。 矩阵乘法：操作 2，t 行非零元素已按行优先顺序给出，矩阵中非零元素的表示为 x y v，其中 x 表示行序号，y 表示列序号，v 表示非零元素值，行列序号从 1 开始。设输入的矩阵为 Q，若 $P \times Q$ 运算合法，则将 $P \times Q$ 的结果矩阵赋给 P，若不合法，则将 Q 赋给 P，同时输出 -1。 矩阵加法：操作 3，t 行非零元素已按行优先顺序给出，矩阵中非零元素的表示为 x y v，其中 x 表示行序号，y 表示列序号，v 表示非零元素值，行列序号从 1 开始。设输入的矩阵为 Q，若 $P+Q$ 运算合法，则将 $P+Q$ 的结果矩阵赋给 P，若不合法，则将 Q 赋给 P，同时输出 -1。 输出操作：操作 4，设当前矩阵 P 的尺寸为 n 行 m 列，第一行输出矩阵 P 的行数和列数，随后 n 行按行优先顺序输出矩阵 P，每行 m 个数字，来表示当前的矩阵内容，每行数字之间用空格分隔。 转置操作：操作 5，设当前矩阵 P 的尺寸为 n 行 m 列，将其转置为 m 行 n 列的矩阵，无需输出。 输入输出格式： 输入： 第一行一个 w 代表操作个数，接下来若干行是各个操作，其中保证第一个操作一定为重置矩阵。 输出： 当执行操作 4 时，输出矩阵 P；当执行操作 2 或 3 时，若对应运算不合法，则输出 -1。		
2. 数据结构与算法描述 （整体思路描述，所需要的数据结构与算法） (1) 本次实验用到的数据结构是稀疏矩阵，当矩阵中的 0 元素非常多的时候，我们使用稀疏矩阵来保存矩阵的信息。稀疏矩阵结构如下：		

```

template<class T>
class sparseMatrix
{
    int rows;
    int cols;
    arrayList<MatrixTerm<T> >array;
public:
    sparseMatrix();
    void reset();
    void output();
    void add();
    void transpose();
    void multi();
    void change_rows_cols(int to_rows,int to_cols);
};

```

(2) 转置操作：对于题目要求的转置操作，我们需要把第一列的每一个非 0 元素转换为第一行的非 0 元素，即为原来在矩阵中的 (i, j) 非 0 元素，变成新的矩阵中的 (j, i) 的非 0 元素，且元素值保持不变。那么我们第一反应就是 for 循环搜索，其中第 i 次搜索，收集第 i 列的元素。但是这样时间复杂度就变成了 $O(\text{cols} * n)$ ，其中 n 是非 0 元素个数，显然时间复杂度过高了。于是我们又想到了，其实第一轮搜索结束后，我们就知道了每个元素的列号，如果我们将它们的相关信息保存下来，是不是就可以降低时间复杂度了？于是想到了用数组保存每一列的非 0 元素在 arrayList 中的下标，这样在转换时候可以 $O(1)$ 访问，非常快捷。于是在收集完下标之后，我们就可以 $O(n)$ 时间复杂度的转换了。代码如下：

```

template<class T>
void sparseMatrix<T>::transpose()
{
    vector<vector<int>>>next( n: cols + 1); //如果到cols那么多开一个
    for (int i = 0; i < array.size(); i++)
    {
        next[array[i].col].push_back(i); //next[i]保存的是第i列的每一个元素在array数组中的id
    }
    //防止被破坏,先收集到vector里边
    vector<MatrixTerm<T>>result;
    for (int i = 1; i <= cols; i++) //收集vector并且回写
    { //next[i][j]存放的是原来的矩阵第i列的非0元素在array中的id
        for (int j = 0; j < next[i].size(); j++)
        {
            result.push_back(MatrixTerm<T>(i, array[next[i][j]].row, array[next[i][j]].val));
        }
    }
    for (int i = 0; i < result.size(); i++)
    {
        array[i] = result[i];
    }
    std::swap( &: rows, &: cols);
}

```

(3) Add 操作:首先要分成两种情况讨论,一个是可以相加,一个是不可以相加。可以相加的需要保存最后的结果并赋值给*this,不可以相加的需要将新输入的矩阵赋值给*this。对于可以相加的,我们结合归并排序的思想,将矩阵映射到一个一维数组,以到原点的距离大小关系为判断依据,哪个到原点的距离小我们就在 result 向量中 Push 进去,如果相等,那么需要判断一下相加是否为 0,如果是,那么 Push。如果不是,那么我们什么都不执行,最后更新私有变量。

```

template<class T>
void sparseMatrix<T>::add()
{
    int new_rows;
    int new_cols;
    int t; //非0元素个数
    cin >> new_rows >> new_cols >> t;

    vector<MatrixTerm<T>> temp; //记录输入的数据
    for (int i = 0; i < t; i++)
    {
        int row, col, val;
        cin >> row >> col >> val;
        temp.push_back(MatrixTerm<T>(row, col, val));
    }
    if (new_rows != rows || new_cols != cols)
    { //更新私有成员
        delete []array.element;
        array.element = new MatrixTerm<T>[t];
        array.arrayLength = t;
        array.listSize = t;
        for (int i = 0; i < t; i++)

```

```

        array[i] = temp[i];
    }
    cout << -1 << endl;
    rows=new_rows;
    cols=new_cols;
}
else
{
    //归并
    vector<MatrixTerm<T> >result;
    auto array_iter = array.begin();
    //vector<MatrixTerm<T> >temp;//记录输入的数据
    auto temp_iter = temp.begin();
    //类似归并排序
    while (array_iter != array.end() && temp_iter != temp.end())
    {
        int array_index = ((*array_iter).row - 1) * cols + (*array_iter).col - 1;//记录到原点的距离
        int temp_index = ((*temp_iter).row - 1) * new_cols + (*temp_iter).col - 1;
        if (array_index < temp_index)
        {
            result.push_back(MatrixTerm<T>(*array_iter->row, array_iter->col, array_iter->val));
            array_iter++;
        }
        else if (array_index == temp_index)
        {
            if (array_iter->val + (*temp_iter).val != 0)//需要特判相加是否等于0
            {
                result.push_back(MatrixTerm<T>(*array_iter->row, array_iter->col, val: array_iter->val + (*temp_iter).val));
            }
            array_iter++;
            temp_iter++;
        }
        else
        {
            result.push_back(MatrixTerm<T>((*temp_iter).row, (*temp_iter).col, (*temp_iter).val));
            temp_iter++;
        }
    }
}

```

```

while (array_iter != array.end())
{
    result.push_back(MatrixTerm<T>(array_iter->row, array_iter->col, array_iter->val));
    array_iter++;
}

while (temp_iter != temp.end())
{
    result.push_back(MatrixTerm<T>((*temp_iter).row, (*temp_iter).col, (*temp_iter).val));
    temp_iter++;
}

//回写,并且更新私有变量
delete [] array.element;
array.listSize=result.size();
array.arrayLength=result.size();
array.element=new MatrixTerm<T>[result.size()];
for (int i = 0; i < result.size(); i++)
{
    array[i] = result[i];
}

```

- (4) Multi 操作: 对于两个稀疏矩阵相乘, 其实类似转置操作, 我们首先需要判断是否可以相乘 (第一个矩阵的列数等于第二个矩阵的行数), 如果可以, 那么进行下一个操作。A 矩阵的第 i 行向量乘以 B 矩阵的第 j 列, 得到的为结果矩阵的第 (i, j) 个元素。同样的, 我们需要记录一下第 i 行的非 0 元素在 array 数组中的第几个, 然后进行相乘, 最后求和。类似与归并排序

```

template<class T>
void sparseMatrix<T>::multi()
{
    //前期准备
    int new_rows;
    int new_cols;
    int t;//新矩阵非0元素的个数
    cin >> new_rows >> new_cols >> t;
    vector<MatrixTerm<T>>temp;//新的矩阵
    for (int i = 0; i < t; i++)
    {
        int row, col, val;
        cin >> row >> col >> val;
        temp.push_back(MatrixTerm<T>(row, col, val));
    }
}

```

```
if (cols != new_rows)//不能相乘
{
    delete[]array.element;
    array.listSize = t;
    array.arrayLength = t;
    array.element = new MatrixTerm<T>[t];
    for (int i = 0; i < t; i++)
    {
        array[i] = temp[i];
    }
    rows=new_rows;
    cols=new_cols;
    cout << -1 << endl;
}
```

```

else
{
    //temp是新的矩阵
    vector<vector<int>>next( n: new_cols + 1);
    vector<MatrixTerm<int>>result;
    for (int i = 0; i < t; i++)
    {
        next[temp[i].col].push_back(i); //按照行优先的顺序输入了id
    }
    int first = 0;
    int last = 0;
    for (int i = 1; i <= rows; i++) //遍历每一行
    {
        //定位原矩阵的首尾
        //i是想要计算的行
        if (array[first].row > i) //先判断是否存在第i行 这里有问题
        {
            continue;
        }
        //一定存在第i行
        while (last < array.size() && array[last].row == i) //防止短路
        {
            last++;
        }
    }
}

```

```
last = last - 1; //回到尾节点
for(int j=1; j<=new_cols; j++)
{
    //类似归并
    int ori_pos = first;
    int new_pos = 0; //next[i]下标
    //矩阵乘法
    int ans=0;
    while (ori_pos <= last && new_pos < next[j].size()) //有问题
    {
        if (array[ori_pos].col < temp[next[j][new_pos]].row)
        {
            ori_pos++;
        }
        else if (array[ori_pos].col == temp[next[j][new_pos]].row)
        {
            ans+= array[ori_pos].val * temp[next[j][new_pos]].val;
            ori_pos++;
            new_pos++;
        }
        else
        {
            new_pos++;
        }
    }
}
```



```

        if(ans!=0)
        {
            result.push_back(MatrixTerm<int>(i,j,ans));
        }
    }
    last++;
    first = last;
}
array.arrayLength=result.size();
array.listSize=result.size();
delete []array.element;
array.element=new MatrixTerm<T>[result.size()];
this->change_rows_cols(rows,new_cols);
for(int i=0;i<result.size();i++)
{
    array[i]=result[i];
}
}
}

```

(5) Output 操作:对于 output 操作,我们只需要判断一下输出的是不是保存的 array 数组中的第一个元素就行,如果是那么直接输出迭代器指向的元素,如果不是,那么输出 0。

```

// T是保存的数据类型
template<class T>
void sparseMatrix<T>::output()
{
    cout << rows << " " << cols << endl;
    typename arrayList<MatrixTerm<T> >::iterator iter = array.begin();
    for (int i = 1; i <= rows; i++)
    {
        for (int j = 1; j <= cols; j++)
        {
            T val = (*iter).row == i && (*iter).col == j ? (*(iter++)).val : 0;
            cout << val << " ";
        }
        cout << endl;
    }
}

```

(6) Reset 操作: 对于 reset 操作我们只需要记录一下非 0 元素的信息即可。然后将非 0 元素组成的数组回写到 array 中去, 并且记得更新私有变量。

```

template<class T>
void sparseMatrix<T>::reset()
{
    int to_rows;
    int to_cols;
    cin >> to_rows >> to_cols;
    vector<MatrixTerm<T> >temp;
    for (int i = 1; i <= to_rows; i++)
    {
        for (int j = 1; j <= to_cols; j++)
        {
            T val;
            cin >> val;
            if (val != 0)
            {
                temp.push_back(MatrixTerm<T>(i, j, val)); //压进去
            }
        }
    }
    delete [] array.element;
    array.listSize = temp.size();
    array.arrayLength = temp.size();
    array.element = new MatrixTerm<T>[temp.size()];

    for (int i = 0; i < temp.size(); i++)
    { //一个一个读出来
        array[i] = temp[i];
    }
    rows = to_rows;
    cols = to_cols;
}

```

3. 测试结果（测试输入，测试输出）

输入：

输入

```
7
1
5 5
2 1 0 0 0
0 0 -1 0 0
0 0 0 0 0
0 0 -1 0 0
0 0 0 0 0
3
5 5
4
2 2 5
3 5 8
4 4 2
5 3 4
4
2
5 5
3
1 1 8
2 4 4
3 5 2
4
5
4
```

输出：

```
运行: untitled94 x
4
5
4 5 5
2 1 0 0 0
0 5 -1 0 0
0 0 0 0 8
0 0 -1 2 0
0 0 4 0 0
5 5
16 0 0 4 0
0 0 0 20 -2
0 0 0 0 0
0 0 0 0 -2
0 0 0 0 8

5 5
16 0 0 0 0
0 0 0 0 0
0 0 0 0 0
4 20 0 0 0
0 -2 0 -2 8

进程已结束，退出代码为 0
```

提交 OJ 最后的结果:

✓ Accepted

#	Result	Score	Time
1	✓ Accepted	5	1 ms
2	✓ Accepted	5	11 ms
3	✓ Accepted	5	55 ms
4	✓ Accepted	5	274 ms
5	✓ Accepted	5	302 ms
6	✓ Accepted	5	1 ms
7	✓ Accepted	5	11 ms
8	✓ Accepted	5	62 ms
9	✓ Accepted	5	329 ms
10	✓ Accepted	5	249 ms
11	✓ Accepted	5	1 ms

4. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）

- (1) 值得注意的是，一定一定要记得更新私有成员，因为 public 接口的函数是没有记忆性的，只有靠着私有成员才能判断自身的状态。并且更新私有成员尽量在成员函数的末尾进行，以防止不必要的思维定式导致的 bug，比如说第一次提交的时候，在 multi 函数中忘记更新不能相乘时候的私有变量的值，导致了错误。经过 debug 发现后才一次 AC，就是下面这个图

```
if (cols != new_rows)//不能相乘
{
    delete[]array.element;
    array.listSize = t;
    array.arrayLength = t;
    array.element = new MatrixTerm<T>[t];
    for (int i = 0; i < t; i++)
    {
        array[i] = temp[i];
    }
    rows=new_rows;
    cols=new_cols;
    cout << -1 << endl;
```

- (2) 想要记录每一列的非 0 元素在原来 array 数组中的下标，那么我们需要开的数组大小是矩阵的列数 +1，而不是列数，否则会 RE
- (3) 学会了二维 vector 的创建，即为：vector 保存的每一个元素都是一个 vector，比如在记录每一列的非 0 元素的时候开的这么一个 vector

```
vector<vector<int>>>next( n: new_cols + 1);
```

- (4) 进行矩阵相乘的时候，我们考虑的应该全面。比如说，我们进入矩阵的时候，可能按照行优先保存的第一个的非 0 元素就是第 4 行的，那么这个时候我们就应该特判，如果不是我们当前正在进行的行，就 continue。同时，我们还需要定位一下这一行的最后一个元素的位置，以达到 O(1) 判断是否到了终点。
- (5) 我们应该有防止短路的意识，尤其是 while 循环的时候，一定要判断是否到了边界。并且判断是否到了边界一定要放在第一位，如果放到了后面很容易短路(尤其是涉及到空指针的情况下)。比如：

```
while (last<array.size() && array[last].row == i)//防止短路
{
    last++;
}
```

空指针造成错误的情况：

```
while (*iter != val && iter != end() )
{
    pre = iter;
    iter++;
}
```

这么写会造成死循环，因为如果这个时候 iter 的值是 nullptr，而 *nullptr 是没有定义的，正确的应该是先判断 iter 是否为 end()，即为：

```
while (iter != end() && *iter != val )
{
    pre = iter;
    iter++;
}
```

- (6) 对于 add 操作，我们需要特判一下相加的最后的值是否为 0 再 push 进去。

- (7) 自己写的时候测的样例都是对的，交到 oj 平台上就 RE 了，怎么办？

解决：RE 常见情况的是数组下标越界，但是经过自己 debug 发现，实际情况是 switch case 条件没有 break 语句，才 RE，在平时，能用 switch case 尽量用 switch case 而不是 If else，因为 switch case 执行的次数少。

5. 附录：实现源代码（本实验的全部源程序代码，程序风格清晰易理解，有充分的注释）

```
1. #include <iostream>
2. #include <vector>
```

```

3.  using namespace std;
4.
5.  template<class T>
6.
7.  class arrayList
8.  {
9.  public:
10.
11.     arrayList(int initialCapacity = 10);
12.     arrayList(const arrayList<T>&);
13.     ~arrayList() { delete[] element; }
14.
15.     T& operator[](int index);
16.     // ADT methods
17.     int size() const { return listSize; }
18.
19.     class iterator;
20.     iterator begin() { return iterator(element); }
21.     iterator end() { return iterator(element + listSize); }
22.
23.
24.     class iterator
25.     {
26.     public:
27.
28.         // constructor
29.         iterator(T* thePosition = 0) { position = thePosition; }
30.
31.         // dereferencing operators
32.         T& operator*() const { return *position; }
33.         T* operator->() const { return &*position; }
34.
35.
36.         // increment
37.         iterator& operator++() // preincrement
38.         {
39.             ++position; return *this;
40.         }
41.         iterator operator++(int) // postincrement
42.         {
43.             iterator old = *this;
44.             ++position;
45.             return old;
46.         }
47.
48.         // equality testing

```

```

49.     bool operator!=(const iterator right) const
50.     {
51.         return position != right.position;
52.     }
53.     bool operator==(const iterator right) const
54.     {
55.         return position == right.position;
56.     }
57.     protected:
58.         T* position;
59.     }; // end of iterator class
60.
61. public: // additional members of arrayList
62.
63.     T* element;        // 1D array to hold list elements
64.     int arrayLength;    // capacity of the 1D array
65.     int listSize;       // number of elements in list
66. };
67.
68. template<class T>
69. arrayList<T>::arrayList(int initialCapacity)
70. { // Constructor.
71.
72.     arrayLength = initialCapacity;
73.     element = new T[arrayLength];
74.     listSize = 0;
75. }
76.
77. template<class T>
78. arrayList<T>::arrayList(const arrayList<T>& theList)
79. { // Copy constructor.
80.     arrayLength = theList.arrayLength;
81.     listSize = theList.listSize;
82.     element = new T[arrayLength];
83.     copy(theList.element, theList.element + listSize, element);
84. }
85.
86.
87. template<class T>
88. T& arrayList<T>::operator[](int index)
89. {
90.     return element[index];
91. }
92.
93.
94. template<class T>

```



```

95. struct MatrixTerm {
96. public:
97.     MatrixTerm<T>(int i,int j,T val):row(i),col(j),val(val){ }
98.     int row;
99.     int col;
100.    T val;
101.    MatrixTerm<T>(){}
102. };
103.
104. template<class T>
105. class sparseMatrix
106. {
107.     int rows;
108.     int cols;
109.     arrayList<MatrixTerm<T> >array;
110. public:
111.     sparseMatrix();
112.     void reset();
113.     void output();
114.     void add();
115.     void transpose();
116.     void multi();
117.     void change_rows_cols(int to_rows,int to_cols);
118. };
119.
120. template<class T>
121. void sparseMatrix<T>::reset()
122. {
123.     int to_rows;
124.     int to_cols;
125.     cin >> to_rows >> to_cols;
126.     vector<MatrixTerm<T> >temp;
127.     for (int i = 1; i <= to_rows; i++)
128.     {
129.         for (int j = 1; j <= to_cols; j++)
130.         {
131.             T val;
132.             cin >> val;
133.             if (val != 0)
134.             {
135.                 temp.push_back(MatrixTerm<T>(i, j, val)); //压进去
136.             }
137.         }
138.     }
139.     delete [] array.element;
140.     array.listSize = temp.size();

```

```

141. array.arrayLength = temp.size();
142. array.element = new MatrixTerm<T>[temp.size()];
143. for (int i = 0; i < temp.size(); i++)
144.     { // 一个一个读出来
145.         array[i] = temp[i];
146.     }
147. rows = to_rows;
148. cols = to_cols;
149. }
150.
151. // T 是保存的数据类型
152. template<class T>
153. void sparseMatrix<T>::output()
154. {
155.     cout << rows << " " << cols << endl;
156.     typename arrayList<MatrixTerm<T> >::iterator iter = array.begin();
157.     for (int i = 1; i <= rows; i++)
158.     {
159.         for (int j = 1; j <= cols; j++)
160.         {
161.             T val = (*iter).row == i && (*iter).col == j ? (*iter++).val : 0;
162.             cout << val << " ";
163.         }
164.         cout << endl;
165.     }
166. }
167.
168. template<class T>
169. void sparseMatrix<T>::add()
170. {
171.     int new_rows;
172.     int new_cols;
173.     int t; // 非 0 元素个数
174.     cin >> new_rows >> new_cols >> t;
175.
176.     vector<MatrixTerm<T> >temp; // 记录输入的数据
177.     for (int i = 0; i < t; i++)
178.     {
179.         int row, col, val;
180.         cin >> row >> col >> val;
181.         temp.push_back(MatrixTerm<T>(row, col, val));
182.     }
183.     if (new_rows != rows || new_cols != cols)
184.     { // 更新私有成员
185.         delete []array.element;
186.         array.element = new MatrixTerm<T>[t];

```

```
187.     array.arrayLength = t;
188.     array.listSize = t;
189.     for (int i = 0; i < t; i++)
190.     {
191.         array[i] = temp[i];
192.     }
193.     cout << -1 << endl;
194.     rows=new_rows;
195.     cols=new_cols;
196. }
197. else
198. {
199.     //归并
200.     vector<MatrixTerm<T> >result;
201.     auto array_iter = array.begin();
202.     //vector<MatrixTerm<T> >temp;//记录输入的数据
203.     auto temp_iter = temp.begin();
204.     //类似归并排序
205.     while (array_iter != array.end() && temp_iter != temp.end())
206.     {
207.         int array_index = ((*array_iter).row - 1) * cols + (*array_iter).col - 1;//记录到原点的距离
208.         int temp_index = ((*temp_iter).row - 1) * new_cols + (*temp_iter).col - 1;
209.         if (array_index < temp_index)
210.         {
211.             result.push_back(MatrixTerm<T>(array_iter->row, array_iter->col, array_iter->val));
212.             array_iter++;
213.         }
214.         else if (array_index == temp_index)
215.         {
216.             if (array_iter->val + (*temp_iter).val != 0)//需要特判相加是否等于 0
217.             {
218.                 result.push_back(MatrixTerm<T>(array_iter->row, array_iter->col, array_iter->val + (*temp_iter)
                .val));
219.             }
220.             array_iter++;
221.             temp_iter++;
222.         }
223.         else
224.         {
225.             result.push_back(MatrixTerm<T>((*temp_iter).row, (*temp_iter).col, (*temp_iter).val));
226.             temp_iter++;
227.         }
228.     }
229.     while (array_iter != array.end())
230.     {
231.         result.push_back(MatrixTerm<T>(array_iter->row, array_iter->col, array_iter->val));
```

```

232.     array_iter++;
233. }
234.
235. while (temp_iter != temp.end())
236. {
237.     result.push_back(MatrixTerm<T>((*temp_iter).row, (*temp_iter).col, (*temp_iter).val));
238.     temp_iter++;
239. }
240.
241. //回写,并且更新私有变量
242. delete [] array.element;
243. array.listSize=result.size();
244. array.arrayLength=result.size();
245. array.element=new MatrixTerm<T>[result.size()];
246. for (int i = 0; i < result.size(); i++)
247. {
248.     array[i] = result[i];
249. }
250.
251. }
252. }
253.
254. template<class T>
255. void sparseMatrix<T>::transpose()
256. {
257.     vector<vector<int>>>next(cols + 1);//如果到 cols 那么需要多开一个
258.     for (int i = 0; i < array.size(); i++)
259.     {
260.         next[array[i].col].push_back(i);//next[i]保存的是第 i 列的每一个元素在 array 数组中的 id
261.     }
262.     //防止被破坏,先收集到 vector 里边
263.     vector<MatrixTerm<T>>result;
264.     for (int i = 1; i <= cols; i++)//收集 vector 并且回写
265.     { //next[i][j]存放的是原来的矩阵第 i 列的非 0 元素在 array 中的 id
266.         for (int j = 0; j < next[i].size(); j++)
267.         {
268.             result.push_back(MatrixTerm<T>(i,array[next[i][j]].row,array[next[i][j]].val));
269.         }
270.     }
271.     for (int i = 0; i < result.size(); i++)
272.     {
273.         array[i] = result[i];
274.     }
275.     std::swap(rows,cols);
276. }
277.

```

```

278. template<class T>
279. void sparseMatrix<T>::multi()
280. {
281.     //前期准备
282.     int new_rows;
283.     int new_cols;
284.     int t;//新矩阵非 0 元素的个数
285.     cin >> new_rows >> new_cols >> t;
286.     vector<MatrixTerm<T>>temp;//新的矩阵
287.     for (int i = 0; i < t; i++)
288.     {
289.         int row, col, val;
290.         cin >> row >> col >> val;
291.         temp.push_back(MatrixTerm<T>(row, col, val));
292.     }
293.     if (cols != new_rows)//不能相乘
294.     {
295.         delete[]array.element;
296.         array.listSize = t;
297.         array.arrayLength = t;
298.         array.element = new MatrixTerm<T>[t];
299.         for (int i = 0; i < t; i++)
300.         {
301.             array[i] = temp[i];
302.         }
303.         rows=new_rows;
304.         cols=new_cols;
305.         cout << -1 << endl;
306.
307.     }
308.     else
309.     {
310.         //temp 是新的矩阵
311.         vector<vector<int>>next(new_cols + 1);
312.         vector<MatrixTerm<int>>result;
313.         for (int i = 0; i < t; i++)
314.         {
315.             next[temp[i].col].push_back(i);//按照行优先的顺序输入了 id
316.         }
317.         int first = 0;
318.         int last = 0;
319.         for (int i = 1; i <= rows; i++)//遍历每一行
320.         {
321.             //定位原矩阵的首尾
322.             //i 是想要计算的行
323.             if (array[first].row > i)//先判断是否存在第 i 行 这里有问题

```

```

324.     {
325.         continue;
326.     }
327.     //一定存在第 i 行
328.     while (last<array.size() &&array[last].row == i)//防止短路
329.     {
330.         last++;
331.     }
332.     last = last - 1;//回到尾节点
333.     for(int j=1;j<=new_cols;j++)
334.     {
335.         //类似归并
336.         int ori_pos = first;
337.         int new_pos = 0;//next[j] 下标
338.         //矩阵乘法
339.         int ans=0;
340.         while (ori_pos <= last && new_pos < next[j].size())//有问题
341.         {
342.             if (array[ori_pos].col < temp[next[j]][new_pos].row)
343.             {
344.                 ori_pos++;
345.             }
346.             else if (array[ori_pos].col == temp[next[j]][new_pos].row)
347.             {
348.                 ans+= array[ori_pos].val * temp[next[j]][new_pos].val;
349.                 ori_pos++;
350.                 new_pos++;
351.             }
352.             else
353.             {
354.                 new_pos++;
355.             }
356.         }
357.         if(ans!=0)
358.         {
359.             result.push_back(MatrixTerm<int>(i,j,ans));
360.         }
361.     }
362.     last++;
363.     first = last;
364. }
365. array.arrayLength=result.size();
366. array.listSize=result.size();
367. delete []array.element;
368. array.element=new MatrixTerm<T>[result.size()];
369. this->change_rows_cols(rows,new_cols);

```

```
370.     for(int i=0;i<result.size();i++)
371.     {
372.         array[i]=result[i];
373.     }
374. }
375. }
376.
377. template<class T>
378. sparseMatrix<T>::sparseMatrix() {}
379.
380. template<class T>
381. void sparseMatrix<T>::change_rows_cols(int to_rows, int to_cols)
382. {
383.     rows=to_rows;
384.     cols=to_cols;
385. }
386.
387.
388. int main()
389. {
390.     int w;
391.     cin.tie(0);
392.     cout.tie(0);
393.     cin >> w;
394.     sparseMatrix<int>matrix;
395.     for (int i = 0; i < w; i++)
396.     {
397.         int flag;
398.         cin >> flag;
399.         switch (flag) {
400.             case 1:
401.                 matrix.reset();
402.                 break;
403.             case 2:
404.                 matrix.multi();
405.                 break;
406.             case 3:
407.                 matrix.add();
408.                 break;
409.             case 4:
410.                 matrix.output();
411.                 break;
412.             case 5:
413.                 matrix.transpose();
414.                 break;
415.         }
```

```
416.  }  
417.  return 0;  
418. }
```