矩阵基本操作的 C++代码

矩阵类

```
// 矩阵类的构造函数
template <class T>
Matrix<T>::Matrix(int r, int c)
{
	if(r<=0||c<=0)                                  // 考察矩阵下标是否合法
	{
		cout<<"bad initializer"<<endl;
		exit(1);
	}
	if(r==1&&c==1)                                  // 规定 1×1 的矩阵不合法
	{
		cout<<"bad initializer"<<endl;
		exit(1);
	}
	rows=r;
	cols=c;
	element=new T [r*c];                            // 为矩阵申请空间
};

// 矩阵类的复制构造函数
template<class T>
Matrix<T>::Matrix(const Matrix<T>&m)
{
	rows=m.rows;
	cols=m.cols;
	element=new T [rows*cols];              // 申请矩阵空间
	for(int i=0; i<rows*cols; i++)                  // 复制矩阵元素
		element[i]=m.element[i];
};

// 重载下标运算符
template<class T>
T & Matrix<T>:: operator () (int i, int j) const
{
	if(i<1||i>rows||j<1||j>cols)
	{
		cout<<"Out Of Bounds"<<endl;
		exit(1);
	}
	return element[(i-1)*cols+j-1];
};

// 重载赋值运算符 =
template<class T>
Matrix<T>& Matrix<T> :: operator = (const Matrix<T>&m)
{
	// 判断是否自我赋值
	if(this!=m )
	{
		rows=m.rows;
		cols=m.cols;
		delete [] element;                      // 释放原矩阵空间
		element=new T [rows*cols];              // 申请新空间
		for(int i=0; i<rows*cols; i++)          // 矩阵赋值
			element[i]=m.element[i];
	}
	return *this ;
};

// 重载二元减法运算符
```

```
template<class T>
Matrix<T> Matrix<T>:: operator - (const Matrix<T>&m) const
{
    if(rows!=m.rows||cols!=m.cols)
    {
        cout<<"Size not matched"<<endl;
        exit(1);
    }
    // 创建一个临时矩阵，存放二矩阵相减的结果
    Matrix<T> w (rows, cols);
    for(int i=0 ; i<rows*cols; i++)
        w.element[i]=element[i]-m.element[i];
        return w;
};

    // 重载乘法操作符
    template<class T>
    Matrix<T> Matrix<T>:: operator * ( const Matrix<T>& m) const {
        if ( cols != m.rows ) {cout <<“Size not matched”<<endl; exit(1);}
        Matrix<T> w (rows, m.cols); // 创建一个临时矩阵，存放二矩阵相乘的结果
        int ct = 0, cm = 0, cw = 0;    // 设定初始位置
        for ( int i =1 ; i <= rows; i++ ){
            for ( int j = 1 ; j <= m.cols ; j++ ){
                T sum = element[ct]*m.element[cm] ;
                for ( int k = 2 ; k <= cols ; k++ ){
                    ct++; // 指向*this 第 i 行的下一个元素
                    cm += m.cols; // 指向 m 第 j 列的下一个元素
                    sum += element[ct] * m.element[cm];
                    }
                w.element[cw] = sum; // 保存计算得到的 w(i,j)值
                cw++;
                ct – = cols–1;    // 重新指向本行的行首元素
                cm = j;           // 指向 m 第 j+1 列的列首元素
                }
            ct += cols;           // 指向下一行的行首元素
            cm = 0;               // 重新指向第一列的列首元素
            }
        return w;
    }
```

三元组表

```
// 构造函数，建立一个 Mrows 行 Mcols 列的稀疏矩阵
template<class T>
SparseMatrix<T>::SparseMatrix(int Mrows, int Mcols)
{
    if(Mrows<=0||Mcols<=0)                    // 考察矩阵下标是否合法
    {
        cout<<"bad initializer"<<endl;
        exit(1);
    }
    if(Mrows==1&&Mcols==1)                // 规定 1×1 的矩阵不合法
    {
        cout<<"bad initializer"<<endl;
        exit(1);
    }
    Rows=Mrows;
    Cols=Mcols;
    Count=0;
    MaxTerm=Mrows *Mcols;
    smArray=new Trituple<T> [Mmaxterm];       // 为矩阵申请空间
};

// 构造函数，建立一个 Mrows 行 Mcols 列 Mcount 个非零元素的稀疏矩阵
```

```
template<class T>
SparseMatrix<T>::SparseMatrix(int Mrows, int Mcols, int Mcount, int Mmaxterm)
{
	if(Mrows<=0||Mcols<=0)                          // 考察矩阵下标是否合法
	{
		cout<<"bad initializer"<<endl;
		exit(1);
	}
	if(Mrows==1&&Mcols==1)                  // 规定 1×1 的矩阵不合法
	{
		cout<<"bad initializer"<<endl;
		exit(1);
	}
	Rows=Mrows;
	Cols=Mcols;
	Count=Mcount;
	MaxTerm=Mmaxterm;
	smArray=new Trituple<T> [Mmaxterm];       // 为矩阵申请空间
};

// 求矩阵的和
template<class T>
SparseMatrix<T> SparseMatrix<T>::Add(SparseMatrix<T> sm)
{
	if((Count+sm.Count)==0)            // 若无非零元素
		return sm;
	SparseMatrix<T> temp(Cols, Rows, 0, Count+sm.Count); //声明一个稀疏矩阵 temp，其行数等
	int k1, k2;                                                    //于原矩阵的列数，其列数等于原矩阵
	k1=k2=0;                                                       //的行数，其非零元素个数初始化为 0
	while((k1<Count)&&(k2<sm.Count))
	{
		int a, b, c, d;
		T e, f;
		smArray[k1].get(a, c, e);
		sm.smArray[k2].get(b, d, f);
		if((a==b)&&(c==d))
		{
			temp.smArray[temp.Count].set(a, c, e+f);
			k1++;
			k2++;
		}
		else
		{
			if((a<b)||((a==b)&&(c<d)))
			{
				temp.smArray[temp.Count].set(a, c, e);
				k1++;
			}
			else
			{
				temp.smArray[temp.Count].set(b, d, f);
				k2++;
			}
		}
		temp.Count++;
	}
	if(k1==Count)
	{
		while(k2<sm.Count)
		{
			int a, b;
			T c;
			sm.smArray[k2].get(a, b, c);
			temp.smArray[temp.Count].set(a, b, c);
			k2++;
			temp.Count++;
```

```
            }
        }
        else
        {
            while(k1<Count)
            {
                int a, b;
                T c;
                smArray[k1].get(a, b, c);
                temp.smArray[temp.Count].set(a, b, c);
                k1++;
                temp.Count++;
            }
        }
        return temp;
};
//求转置矩阵
template<class T>
SparseMatrix<T> SparseMatrix<T> :: Transpose（）
{
        SparseMatrix<T> b ;                                    //   声明一个稀疏矩阵 b
        b.Rows = Cols ;                                        //   b 的行数等于原矩阵的列数
        b.Cols = Rows ;                                        //   b 的列数等于原矩阵的行数
        b.Count = Count ;                                      //   b 与原矩阵的非零元素个数相同
        if（ Count > 0 ）                                      //   若有非零元素
        {
            int Bnumber = 0 ;
            for（ k = 0 ; k < Cols ; k + + ）                  // 对矩阵 b 按行优先依次确认非零元素
                for（ i = 0 ; i < Count ; i + + ）             // 扫描原矩阵的三元组表
                    if（ smArray[ i ].col = = k ）             // 是否有列号为 k 的非零元素
                    {
                        b.smArray[ Bnumber ].row = k ;
                        b.smArray[ Bnumber ].col = smArray[ i ].row ;
                        b.smArray[ Bnumber ].value = smArray[ i ].value ;
                        Bnumber + + ;
                    }
        }
        return b ;   // 返回转置矩阵 b
};
```

十字链表

```
/* 十字链表类结点 CLNode 的定义 */
template<class T>
class CLNode
{
public:
        int ROW, COL;                   // 数据域
        T VAL;
        CLNode<T> *LEFT, *UP;           // 指针域
        CLNode()  {ROW=COL=-1;LEFT=UP=NULL;}    // 构造函数
        CLNode(int r, int c, T v):ROW(r), COL(c), VAL(v) { LEFT=UP=NULL;}
};

/* 十字链表类 CrossLink 的定义 */
template<class T>
class SparseMatrix_C
{
private:
        int Rows, Cols, Count;          // 稀疏矩阵的行数，列数及非零元素个数
        CLNode<T> **CLROW;              // 行表头数组
        CLNode<T> **CLCOL;              // 列表头数组
public:
        // 构造函数，建立一个 Mrows 行 Mcols 列 Mcount 个非零元素的稀疏矩阵
```

```
        SparseMatrix_C(int Mrows, int Mcols, int Mcount);
        SparseMatrix_C<T>& BaseOperation(int Baserow, int Basecol);      //主步骤操作
        int get_Rows() const{return Rows;}              // 返回矩阵行数
        int get_Cols() const{return Cols;}          // 返回矩阵列数
};
```

算法 SparseMatrix_C

```
// 十字链表类的构造函数
template<class T>
SparseMatrix_C<T>::SparseMatrix_C(int Mrows, int Mcols, int Mcount)
{
        if(Mrows<=0||Mcols<=0)              // 考察矩阵下标是否合法
        {
                cout<<"bad initializer"<<endl;
                exit(1);
        }
        if(Mrows==1&&Mcols==1) // 规定 1×1 的矩阵不合法
        {
                cout<<"bad initializer"<<endl;
                exit(1);
        }
        Rows=Mrows;    Cols=Mcols;        Count=Mcount;
        //申请空间
        CLROW=new CLNode<T>* [Rows];
        CLCOL=new CLNode<T>* [Cols];
        //初始化
        int i;
        for(i=0; i<Rows; i++)
        {
                CLROW[i]=new CLNode<T>( );
                CLROW[i]->LEFT=CLROW[i];
        }
        for(i=0; i<Cols; i++)
        {
                CLCOL[i]=new CLNode<T>( );
                CLCOL[i]->UP=CLCOL[i];
        }
};
```

算法 SP

```
//稀疏矩阵的主步骤操作，稀疏矩阵的表示方式为正交链表
template<class T>
SparseMatrix_C<T>&SparseMatrix_C<T>::BaseOperation(int I0, int J0)
{
        //SP1: 初始化
        CLNode<T> *temp=CLROW[I0]->LEFT;
        while(temp->COL!=J0)
                temp=temp->LEFT;
        const T Alpha=1.0/temp->VAL;
        temp->VAL=Alpha;
        CLNode<T> *P0=CLROW[I0];
        CLNode<T> *Q0=CLCOL[J0];
        CLNode<T> *PTR[100];
        //SP2: 处理主行 I0
        P0=P0->LEFT;
        int J=P0->COL;
        while(J!=-1)
        {
                PTR[J]=CLCOL[J];
                P0->VAL*=Alpha;
                P0=P0->LEFT;
                J=P0->COL;
        }
        //SP3: 找新行 I, 并指定 P1
        while(1)
```

```
    {
        Q0=Q0->UP;
        int I=Q0->ROW;
        if(I<0)
            return *this;
        if(I==I0)
            continue;
        CLNode<T> *P=CLROW[I];
        CLNode<T> *P1=P->LEFT;
        //SP4: 确定新列 J
        do
        {
            P0=P0->LEFT;
            J=P0->COL;
            if(J<0)
                break;
            if(J==J0)
                continue;
            // SP5: P1 所指元素所在的列与 J 列比较
            while(P1->COL>J)
            {
                P=P1;
                P1=P->LEFT;
            }
            if(P1->COL==J)
            {
                //SP7: 主步骤操作
                P1->VAL-=Q0->VAL*P0->VAL;
                if(P1->VAL==0)
                {
                    //SP8: 删除零元素
                    PTR[J]->UP=P1->UP;
                    P->LEFT=P1->LEFT;
                    delete P1;
                    P1=P->LEFT;
                }
                else
                {
                    PTR[J]=P1;
                    P=P1;
                    P1=P->LEFT;
                }
            }
            else
            {
                // SP6: 插入新元素
                CLNode<T> *X=new CLNode<T>;
                X->VAL=-Q0->VAL*P0->VAL;
                X->ROW=I;
                X->COL=J;
                X->LEFT=P1;
                X->UP=PTR[J]->UP;
                P->LEFT=X;
                P=X;
                PTR[J]->UP=X;
                PTR[J]=X;
            }
        }
        while(J>=0);
        Q0->VAL*=-Alpha;
    }
};
```