山东大学 计算机科学与技术 学院

数据结构与算法 课程实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：201700130009 | 姓名： 张愈博 | | 班级： 计科17.3 |
| 实验题目：数组与矩阵 | | | |
| 实验学时：2 | | 实验日期： 2018.11.01 | |
| 实验目的：  1.掌握稀疏矩阵结构的描述及操作的实现。 | | | |
| 软件环境：  Windows10  Dev C++ 5.11 | | | |
| 1.实验内容（题目内容，输入要求，输出要求）  （1）创建稀疏矩阵类，采用行主顺序把稀疏矩阵非 0 元素映射到一维数组中，  提供操作：两个稀疏矩阵相加、两个稀疏矩阵相乘、输出矩阵（以通常的阵列形式输出）。  （2）键盘输入矩阵的行数、列数；按行输入矩阵的各元素值，建立矩阵；  （3）对建立的矩阵执行相加、相乘的操作，输出操作的结果矩阵。  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）稀疏矩阵的元素是含行列数和值的类，并重载了输出<<运算符、>元素安抚和<运算符，稀疏矩阵类中的成员变量有行数row，列数col,非0项Arraylist表terms，成员函数有转置函数、加法函数、乘法函数，阵列输出函数，并重载了输入输出运算符。  （2）加法函数算法：使用矩阵的迭代器，遍历两个矩阵的所有元素，在这个过程中比较两个元素的索引大小，如果索引相同且两个元素相加不为0，则把它们的和存入结果矩阵，如果索引相同，直接把索引小的元素存入结果矩阵。  （3）乘法函数算法：把两个矩阵的元素看成两组数对，遍历第一个矩阵中所有数对，判断其是否可以和第二组数对相乘，如果可以相乘，再判断它们的结果的下标是否已存在于结果矩阵中，若存在则更新已有数据，若不存在则存入结果矩阵。  （4）转置函数算法：首先计算矩阵中每一列的长度，和每一列第一个非0元素在一维数组中的索引。然后通过迭代器把原矩阵中的元素放到转置矩阵的合适位置中。  （5）阵列输出算法：遍历稀疏矩阵中每一个元素，当遇到非0元素后才输出，否则输出0。注意这里使用了不需输出二维数组的节约空间复杂度的方法。  3.测试结果（测试输入，测试输出，结果分析）    图1 图2    图3 图4  4.分析与探讨（结果分析，若存在问题，探讨解决问题的途径）  图1是矩阵A的输入过程，图2是矩阵B的输入过程，图3是加法的结果，图4是乘法的结果。经验算，运算结果均正确。但是在实验的过程中遇到了不少问题，一下罗列一些在今后的实验中需注意的几条：  （1）在加法和乘法函数中，为了不让结果矩阵中的元素数目溢出，均使用arraylist类中的insert函数，这样在数组长度不够时自动拓展大小。  （2）要注意设计类的对外接口，如不能直接引用数组线性表类对象的terms中的element，但是可以通过迭代器实现对它的各种拓展操作。当线性表由链表描述时，使用迭代器的优势会更加明显。而且就私有成员的安全性角度来说，使用迭代器虽然可以对数组中的元素进行引用，却无法进行修改，增加了数据安全性。  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）  Main.cpp  #include<iostream>  #include"稀疏矩阵.h"  #include"Arraylist.h"  using namespace std;  int main()  {  sparseMatrix<int> A,B,C,D;  cin>>A;  cout<<A<<endl;  cin>>B;  cout<<B<<endl<<"加法结果："<<endl;  A.add(B,D);  D.matrix\_print();  C=A\*B;  cout<<"乘法的结果是"<<C<<endl;  C.matrix\_print();  return 0;  }  Arraylist.h  #pragma once  #include<iostream>  template<class T>  class Arraylist  {  public:  Arraylist(int initialCapacity=10);  Arraylist(const Arraylist<T>& A);  ~Arraylist() { delete [] element; }    void reSet(int newSize);  void set(int Index,T ele);  void clear();  bool empty() const  { return listSize==0; }  int size() const  { return listSize; }  T& get(int theIndex) const  {//实现随机存储，效率极高 O(1)  checkIndex(theIndex);  return element[theIndex];  }  int indexOf(T& x) const;  void erase(int theIndex);  void insert(int theIndex,const T& theElement);  void output() const;  int capacity() const  { return arrayLength; }  friend std::ostream& operator<< (std::ostream& out,const Arraylist<T> &A) //模板类的友元函数声明与输出不能分离  {  for(int i=0;i<A.listSize;i++)  { out<<A.element[i]; }  return out;  }  class iterator  {  public:  iterator(T \*postion=0)  {  pos=postion;  }  T& operator\*() const { return \*pos; }  T\* operator->() const { return &\*pos; }  iterator& operator++() { ++pos; return \*this; }  iterator operator++(int) { iterator old=\*this; ++pos; return old; }  bool operator!=(const iterator right) const{ return pos!=right.pos; }  bool operator==(const iterator right) const{ return pos==right.pos; }  protected:  T \*pos;  };  iterator begin()  {  return iterator(element);  }  iterator end()  {  return iterator(element+listSize);  }  int Term\_search(T& x)  {//稀疏矩阵专用search,如果找到了x,则返回它的索引，否则返回-1  //std::cout<<"listSize is "<<listSize<<std::endl;  for(int i=0;i<listSize;i++)  {  if(x.row==element[i].row&&x.col==element[i].col) return i;  }  return -1;  }  void matrixTerm\_sort()  {  for(int i=0;i<listSize;i++)  {  for(int j=i;j<listSize;j++)  {  if(element[i]>element[j])  swap(element[i],element[j]);  else ;  }  }  }  void swap(T &A,T &B)  {  int temp;  temp=A.row; A.row=B.row; B.row=temp;  temp=A.col; A.col=B.col; B.col=temp;  temp=A.value; A.value=B.value; B.value=temp;    }  //protected:  void checkIndex(int theIndex) const  {//确定索引的有效性，防止越界  if(theIndex<0||theIndex>=listSize)  {  std::cerr<<"请检查索引的有效性！"<<std::endl;  }  }  //若索引无效，则抛出异常  T \*element; //存储线性表元素的一维数组  int arrayLength; //一维数组的容量  int listSize; //线性表的元素个数  };  Arraylist.cpp  #pragma once  #include"Arraylist.h"  template<class T>  Arraylist<T>:: Arraylist(int initialCapacity)  {//不考虑智障用户输负数  element=new T [initialCapacity]; //这里已经初始化了一遍。  arrayLength=initialCapacity;  listSize=0;  }  template<class T>  Arraylist<T>:: Arraylist(const Arraylist<T>& A)  {  listSize=A.listSize;  arrayLength=A.arrayLength;  element=new T [arrayLength];  for(int i=0;i<listSize;i++)  {  element[i]=A.element[i];  }  }  template<class T>  int Arraylist<T>:: indexOf(T& x) const  {//返回元素x第一次出现时的索引;若不存在，则返回-1  for(int i=0;i<listSize;i++)  {  if(x==element[i]) return i;  }  return -1;  }  template<class T>  void Arraylist<T>:: erase(int theIndex)  {//从表中删除索引为theIndex的元素  for(int i=theIndex;i<listSize-1;i++)  {  element[i]=element[i+1];  }  element[--listSize].~T(); //调用析构函数。？？verstehen. 单个类对象的析构，并不是动态调用  }  template<class T>  void Arraylist<T>:: insert(int theIndex,const T& theElement)  {  if(theIndex<0||theIndex>listSize)  std::cerr<<"请输入正确索引！"<<std::endl;  //std::cout<<listSize<<" "<<arrayLength<<std::endl;  if(listSize==arrayLength&&arrayLength!=0)  {//数组空间已满  T \*p=new T[2\*arrayLength];  for(int i=0;i<listSize;i++)  {  p[i]=element[i];  }  delete [] element;  element=p;  arrayLength\*=2;  }  if(arrayLength==0)  {//数组为空  element=new T[2];  }  //std::cout<<"listSize is "<<listSize<<std::endl;  for(int i=listSize-1;i>=theIndex;i--)  {  element[i+1]=element[i]; //第二次调用插入函数错误出在这里,注意数组下标是从0开始的  }  element[theIndex]=theElement;  listSize++;  }  template<class T>  void Arraylist<T>:: output() const  {//输出该线性表  for(int i=0;i<listSize;i++)  {  std::cout<<element[i];  }  }  template<class T>  void Arraylist<T>::reSet(int newSize)  {//把数组的元素个数改为newSize  if(arrayLength<newSize)  {//不够的话扩大数组容量  T \*p=new T[newSize];  for(int i=0;i<listSize;i++)  {  p[i]=element[i];  }  delete [] element;  element=p;  arrayLength=newSize;  }  listSize=newSize;  }  template<class T>  void Arraylist<T>::set(int Index,T ele)  {  if(Index<0||Index>listSize)  std::cerr<<"请输入正确索引！"<<std::endl;    else{  element[Index]=ele;  }  }  template<class T>  void Arraylist<T>::clear()  {  delete [] element;  arrayLength=listSize=0;  }  稀疏矩阵.h  #pragma once  #include"Arraylist.cpp"  using namespace std;  template<class T>  class matrixTerm  {  public:  int row,col;  T value;  friend ostream& operator<<(ostream &out,matrixTerm<T> &A)  {  out<<"row="<<A.row<<",col="<<A.col<<",value="<<A.value<<endl;  }  bool operator<(matrixTerm<T> &A)  {  if(row<A.row) return true;  if(row==A.row)  {  if(col<A.col) return true;  else return false;  }  if(row>A.row) return false;  }  bool operator>(matrixTerm<T> &A)  {  if(row<A.row) return false;  if(row==A.row)  {  if(col<A.col) return false;  else return true;  }  if(row>A.row) return true;  }  };  template<class T>  class sparseMatrix  {  public:  void transpose(sparseMatrix<T> &A)  {//把\*this转置后放入A  A.row=row;  A.col=col;  A.terms.reSet(terms.size());    int \*colSize=new int[col+1]; //原矩阵每一列非零元素数量  int \*rowNext=new int[col+1]; //原矩阵每一列第一个非0元素在一维数组中的索引    for(int i=1;i<col;i++)  colSize[i]=0;  for(typename Arraylist < matrixTerm<T> >::iterator i=terms.begin();i!=terms.end();i++)  { colSize[(\*i).col]++; }    rowNext[1]=0;  for(int i=2;i<=col;i++)  {//记录下A中的每行起始点  rowNext[i]=rowNext[i-1]+colSize[i-1];  }    matrixTerm<T> temp;  for(typename Arraylist < matrixTerm<T> >::iterator i=terms.begin();i!=terms.end();i++)  {  int j=rowNext[(\*i).col]++; //j是元素在A表中的位置  temp.row=(\*i).row;  temp.col=(\*i).col;  temp.value=(\*i).value;  A.terms.set(j,temp);  }  }  void add(sparseMatrix<T> &B,sparseMatrix<T> &C)  {//计算C=(\*this)+B  if(row!=B.row||col!=B.col)  std::cerr<<"矩阵不相容！\n";  //设置结果矩阵的特征  C.row=row;  C.col=col;  //C.terms.clear(); 加这个就会报错,出现奇怪的数字  int Csize=0;  //定义\*this和B的迭代器  typename Arraylist < matrixTerm<T> >::iterator it=terms.begin();  typename Arraylist < matrixTerm<T> >::iterator itend=terms.end();  typename Arraylist < matrixTerm<T> >::iterator ib=B.terms.begin();  typename Arraylist < matrixTerm<T> >::iterator ibend=B.terms.end();  //遍历\*this和B，把相关项相加  while(it!=itend&&ib!=ibend)  {  int tIndex=( (\*it).row-1 ) \* col+(\*it).col;  int bIndex=( (\*ib).row-1 ) \* col+(\*ib).col;  //确定索引，然后比较大小  if(tIndex<bIndex)  {//如果前者小  C.terms.insert(Csize++,(\*it));  it++;  }    if(tIndex>bIndex)  {//如果前者大  C.terms.insert(Csize++,(\*ib));  ib++;  }    if(tIndex==bIndex)  {//先看两个矩阵对应元素的和是否等于0  if((\*it).value+(\*ib).value==0)  { it++; ib++; }  else  {  matrixTerm<T> temp;  temp.col=(\*it).col;  temp.row=(\*it).row;  temp.value=(\*it).value+(\*ib).value;  C.terms.insert(Csize++,temp);  it++; ib++;  }  }  }  //处理剩余的元素  for( ;it!=itend;it++)  C.terms.insert(Csize++,(\*it));  for( ;ib!=ibend;ib++)  C.terms.insert(Csize++,(\*ib));    }  friend ostream& operator<<(ostream &out,sparseMatrix<T> &A)  {//重载输出运算符  //首先输出矩阵特征  int size=A.terms.size();  out<<"行数："<<A.row<<"\t列数："<<A.col<<"\t非0元素个数:"<<size<<endl;  for(typename Arraylist< matrixTerm<T> >::iterator i=A.terms.begin();i!=A.terms.end();i++ )  {  out<<"a("<<(\*i).row<<","<<(\*i).col<<")="<<(\*i).value<<"\t";  }  return out;  }  friend istream& operator>>(istream &in,sparseMatrix<T> &A)  {//重载输入运算符，以行主次序输入  //首先输入矩阵特征  int num;  cout<<"请输入该稀疏矩阵的行数、列数和非0元素数量"<<endl;  in>>A.row>>A.col>>num;  if(A.row<=0||A.col<=0||num<0||num>A.row\*A.col)  {//检查合法性  cerr<<"输入数据非法！"<<endl;  }  //设置arraylist term的大小  A.terms.reSet(num);  //输入每一项  matrixTerm<T> temp\_term;  for(int i=1;i<=num;i++)  {  cout<<"请输入第"<<i<<"个元素行号、列号和元素值"<<endl;  in>>temp\_term.row>>temp\_term.col>>temp\_term.value;  if(temp\_term.row<0||temp\_term.row>A.row||temp\_term.col<0||temp\_term.col>A.col)  {//检查合法性  cerr<<"输入数据非法！"<<endl;  }  A.terms.set(i-1,temp\_term);  }  return in;  }  sparseMatrix& operator\*(sparseMatrix<T> &B)  {//稀疏矩阵乘法，将\*this和B相乘返回一个新矩阵  if(col!=B.row)  {  cerr<<"数学错误！"<<endl;  exit(1);  }  //下面是创建临时的结果矩阵 ,并初始化  int i=0,j=0,k=0,rc=0;  sparseMatrix<T> \*Res=new sparseMatrix<T>; //其中的arraylist默认好像只有10个长度  Res->row=row;  Res->col=B.col;    //当\*this和A矩阵中元素都没有处理完  for(i=0;i<terms.size();i++)  {//把\*this中的一个元素拿来和A中对应元素相比较  for(j=0;j<B.terms.size();j++)  {  if(terms.element[i].col==B.terms.element[j].row)  {//发现可以相乘的项  matrixTerm<T> TEMP;  TEMP.row=terms.element[i].row; TEMP.col=B.terms.element[j].col;  TEMP.value=terms.element[i].value\*B.terms.element[j].value;  cout<<TEMP<<endl;  if( Res->terms.Term\_search(TEMP)==-1 )  {//若在已保存的数对中没有下标相同的数对  Res->terms.insert(rc++,TEMP);  }  else if(Res->terms.Term\_search(TEMP)>=0)  { //若已经存在下标相同的数对,把前后的结果相加  int index=Res->terms.Term\_search(TEMP);  Res->terms.element[index].value+=TEMP.value;  }  }  }  }  return \*Res;  }  //稀疏矩阵乘法，输入A,B返回一个新矩阵  void matrix\_print()  {//看我一波操作，不用二维数组！！  int k=0;  for(int i=1;i<=row;i++)  {  for(int j=1;j<=col;j++)  {  if( terms.element[k].row==i && terms.element[k].col==j )  cout<<terms.element[k++].value<<" ";  else cout<<"0 ";  }  cout<<endl;  }  }  private:  int row,col;//矩阵特征  Arraylist < matrixTerm<T> > terms; //非0项表  }; | | | |