**附件：**

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：202000130138 | 姓名：宋璎航 | | 班级： 20.3 |
| 实验题目：数组和矩阵 | | | |
| 实验学时：2 | | 实验日期： 2020/10/26 | |
| 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：  VSC | | | |
| 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PxQ运算合法,则将PxQ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。   1. 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）   采用三元组存储单位矩阵元素，矩阵的加法采用双迭代器法，按照行主矩阵绝对序列索引号（即行号乘总列数+列号）进行迭代。矩阵乘法使用加法计数器每一行一个循环。转置即把每个三元组行列颠倒，再使其按行主排序即可   1. 测试结果（测试输入，测试输出）  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**  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   1. 分析与探讨（结果分析，若存在问题，探讨解决问题的途径）   线性表结构是一种基础结构，没有必要刻意用线性表类来描述，可以只用单独的指针，这样排查错误会比较方便。   1. 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）   *#include* "bits/stdc++.h"  template <class T>  class linearList  {  public:      virtual ~linearList(){};      virtual bool empty() const = 0;      virtual int size() const = 0;      virtual T &get(int theIndex) const = 0;      virtual int indexOf(const T &theElement) const = 0;      virtual void erase(int theIndex) = 0;      virtual void insert(int theIndex, const T &theElement) = 0;  };  template <class T>  class arrayList : public linearList<T>  {      int arrayLength;      bool checkIndex(int theIndex) const;  public:      T \*element;      int listSize;      arrayList();      arrayList(int initialCapacity);      arrayList(const arrayList<T> &);      ~arrayList() { delete[] element; }      bool empty() const { *return* listSize == 0; }      int size() const { *return* listSize; }      T &get(int theIndex) const;      int indexOf(const T &theElement) const;      void erase(int theIndex);      void insert(int theIndex, const T &theElement);      T \*firstP() const { *return* element; }      void reverse();      void merge(const arrayList<T> &a, const arrayList<T> &b);      void output() const;      void clear();  };  template <class T>  void arrayList<T>::clear()  {      delete[] element;      element = new T[arrayLength];      listSize = 0;  }  template <class T>  arrayList<T>::arrayList()  {      arrayLength = 1000000;      element = new T[arrayLength];      listSize = 0;  }  template <class T>  arrayList<T>::arrayList(int initialCapacity)  {  *if* (initialCapacity < 1)          exit(-1);      arrayLength = initialCapacity;      element = new T[arrayLength];      listSize = 0;  }  template <class T>  arrayList<T>::arrayList(const arrayList<T> &theList)  {      arrayLength = theList.arrayLength;      listSize = theList.listSize;      element = new T[arrayLength];      copy(theList.element, theList.element + listSize, element);  }  template <class T>  bool arrayList<T>::checkIndex(int theIndex) const  {  *if* (theIndex < 0 or theIndex >= listSize)  *return* false;  *else*  *return* true;  }  template <class T>  T &arrayList<T>::get(int theIndex) const  {  *if* (checkIndex(theIndex))  *return* element[theIndex];  *else*          std::cout << "theIndex is out of range" << std::endl;  *return* element[0];  }  template <class T>  int arrayList<T>::indexOf(const T &theElement) const  {  *for* (int i = 0; i < listSize; ++i)      {  *if* (element[i] == theElement)  *return* i;      }  *return* -1;  }  template <class T>  void arrayList<T>::erase(int theIndex)  {  *if* (!checkIndex(theIndex))  *return*;      std::copy(element + theIndex + 1, element + listSize, element + theIndex);      element[--listSize].~T();  }  template <class T>  void arrayList<T>::insert(int theIndex, const T &theElement)  {  *if* (theIndex < 0 or theIndex > listSize)      {          std::cout << "theIndex is out of range" << std::endl;  *return*;      }  *if* (listSize == arrayLength)      {          T \*temp = new T[arrayLength];          std::copy(element, element + listSize, temp);          delete[] element;          element = new T[2 \* arrayLength];          std::copy(temp, temp + listSize, element);          delete[] temp;      }      std::copy\_backward(element + theIndex, element + listSize, element + listSize + 1); *//此处不能用copy*      element[theIndex] = theElement;      listSize++;  }  template <class T>  void arrayList<T>::reverse()  {      T \*temp;      temp = new T[listSize];  *for* (int i = 0; i < listSize; ++i)          temp[i] = element[listSize - i - 1];      copy(temp, temp + listSize, element);      delete[] temp;  }  template <class T>  void arrayList<T>::output() const  {  *for* (int i = 0; i < listSize; ++i)          std::cout << element[i] << '\n';  }  template <class T>  void arrayList<T>::merge(const arrayList<T> &a, const arrayList<T> &b)  {      T \*temp;      temp = new T[a.listSize + b.listSize];      int ca = 0, cb = 0;  *while* (ca != a.listSize or cb != b.listSize)      {  *if* (ca == a.listSize)          {              temp[ca + cb] = b.element[cb];              ++cb;  *continue*;          }  *if* (cb == b.listSize)          {              temp[ca + cb] = a.element[ca];              ++ca;  *continue*;          }  *if* (a.element[ca] < b.element[cb])          {              temp[ca + cb] = a.element[ca];              ++ca;          }  *else*          {              temp[ca + cb] = b.element[cb];              ++cb;          }      }      this->element = new T[a.listSize + b.listSize];      this->listSize = a.listSize + b.listSize;      this->arrayLength = this->listSize;      std::copy(temp, temp + a.listSize + b.listSize, this->element);      delete[] temp;  }  template <class T>  struct matrixTerm  {      matrixTerm() : row(0), col(0), value(0) {}      matrixTerm(int theRow, int theCol, T theValue) : row(theRow), col(theCol), value(theValue) {}      int row, col;      T value;  };  template <class T>  bool operator==(matrixTerm<T> a, matrixTerm<T> b)  {  *return* (a.row == b.row and a.col == b.col and a.value == b.value);  }  template <class T>  std::ostream &operator<<(std::ostream &out, matrixTerm<T> &x)  {      out << x.row << ' ' << x.col << ' ' << x.value;  *return* out;  }  template <class T>  class sparseMatrix  {  private:      int rows, cols;      arrayList<matrixTerm<T>> terms;  public:      sparseMatrix() {}      sparseMatrix(int theRow, int theCol) : rows(theRow), cols(theCol) {}  *// sparseMatrix(sparseMatrix<T> &theMatrix);*      ~sparseMatrix(){};      void reset(int theRow, int theCol);      T get(int theRow, int theCol) const;      void set(matrixTerm<T> &theTerm);      void set(matrixTerm<T> &theTerm, int theIndex);      void transpose();      void add(sparseMatrix<T> &b);      void multi(sparseMatrix<T> &b);      void output() const;      sparseMatrix<T> &operator=(const sparseMatrix<T> &theMatrix);  };  template <class T>  void sparseMatrix<T>::reset(int theRow, int theCol)  {      rows = theRow;      cols = theCol;      terms.clear();  }  template <class T>  void sparseMatrix<T>::transpose()  {      sparseMatrix<T> ans(cols, rows);      int \*colSize = new int[cols + 1];      int \*rowNext = new int[cols + 1];  *for* (int i = 1; i <= cols; ++i)          colSize[i] = 0;  *for* (int i = 0; i < terms.size(); ++i)          colSize[terms.element[i].col]++;      rowNext[1] = 0;  *for* (int i = 2; i <= cols; ++i)          rowNext[i] = rowNext[i - 1] + colSize[i - 1];  *for* (int i = 0; i < terms.size(); ++i)      {          int j = rowNext[terms.element[i].col]++;          matrixTerm<T> mt(terms.element[i].col, terms.element[i].row, terms.element[i].value);          ans.set(mt, j);      }      \*this = ans;  }  template <class T>  T sparseMatrix<T>::get(int theRow, int theCol) const  {  *if* (theRow > rows or theCol > cols or theRow <= 0 or theCol <= 0)          exit(-1);      int cnt = 0;  *while* (cnt < terms.size() and (terms.get(cnt).col != theCol or terms.get(cnt).row != theRow))          cnt++;  *if* (cnt == terms.size())  *return* 0;  *else*  *return* terms.get(cnt).value;  }  template <class T>  void sparseMatrix<T>::set(matrixTerm<T> &theTerm)  {  *if* (terms.size() == 0)      {          terms.insert(0, theTerm);  *return*;      }      int theIndex = 0;  *while* (theIndex < terms.size() and terms.get(theIndex).row \* cols + terms.get(theIndex).col < theTerm.row \* cols + theTerm.col)          theIndex++;  *if* (theIndex < terms.size())  *if* (terms.get(theIndex).row == theTerm.row and terms.get(theIndex).col == theTerm.col)  *return*;      terms.insert(theIndex, theTerm);  }  template <class T>  void sparseMatrix<T>::set(matrixTerm<T> &theTerm, int theIndex)  {      terms.element[theIndex] = theTerm;      ++terms.listSize;  }  template <class T>  void sparseMatrix<T>::add(sparseMatrix<T> &b)  {  *if* (rows != b.rows or cols != b.cols)      {          \*this = b;          std::cout << -1 << std::endl;  *return*;      }      sparseMatrix<T> ans(rows, cols);      int cnta = 0, cntb = 0, cntans = 0;  *while* (cnta < terms.size() and cntb < b.terms.size())      {          matrixTerm<T> terma = terms.element[cnta];          matrixTerm<T> termb = b.terms.element[cntb];          int Indexa = terma.row \* cols + terma.col;          int Indexb = termb.row \* cols + termb.col;  *if* (Indexa < Indexb)          {              ans.set(terma, cntans++);              cnta++;  *continue*;          }  *else* *if* (Indexa > Indexb)          {              ans.set(termb, cntans++);              cntb++;  *continue*;          }  *else*          {  *if* (terma.value + termb.value != 0)              {                  matrixTerm<T> termab(terma.row, terma.col, terma.value + termb.value);                  ans.set(termab, cntans++);                  cnta++;                  cntb++;  *continue*;              }  *else*              {                  cnta++;                  cntb++;  *continue*;              }          }      }  *while* (cntb < b.terms.size())          ans.set(b.terms.element[cntb++], cntans++);  *while* (cnta < terms.size())          ans.set(terms.element[cnta++], cntans++);      \*this = ans;  }  template <class T>  void sparseMatrix<T>::multi(sparseMatrix<T> &b)  {  *if* (cols != b.rows)      {          \*this = b;          std::cout << -1 << '\n';  *return*;      }      sparseMatrix<T> ans(rows, b.cols);      int cntans = 0;      int sizeofrow[10000];      int nextofrow[10000];      int theanswer[10000];  *for* (int i = 1; i <= b.rows; ++i)          sizeofrow[i] = 0;  *for* (int i = 0; i < b.terms.size(); ++i)  *if* (b.terms.element[i].value != 0)              sizeofrow[b.terms.element[i].row]++;      nextofrow[1] = 0;  *for* (int i = 2; i <= b.terms.size(); ++i)          nextofrow[i] = nextofrow[i - 1] + sizeofrow[i - 1];      int p = 0;  *for* (int i = 1; i <= rows and p < terms.size(); ++i)      {  *for* (int j = 1; j <= b.cols; ++j)              theanswer[j] = 0;  *while* (p < terms.size() and terms.element[p].row == i)          {              int t = terms.element[p].col;  *if* (sizeofrow[t] != 0)  *for* (int q = nextofrow[t]; q < nextofrow[t] + sizeofrow[t]; ++q)                      theanswer[b.terms.element[q].col] += terms.element[p].value \* b.terms.element[q].value;              ++p;          }  *for* (int k = 1; k <= b.cols; ++k)          {  *if* (theanswer[k] != 0)              {                  matrixTerm<T> mt(i, k, theanswer[k]);                  ans.set(mt, cntans++);              }          }      }      \*this = ans;  }  template <class T>  void sparseMatrix<T>::output() const  {      std::cout << rows << ' ' << cols << '\n';  *// terms.output();*      int cnt = 0;  *for* (int i = 1; i <= rows; ++i)      {  *for* (int j = 1; j <= cols; ++j)          {  *if* (cnt < terms.size() and terms.element[cnt].row == i and terms.element[cnt].col == j)              {                  std::cout << terms.element[cnt].value << ' ';                  ++cnt;              }  *else*                  std::cout << 0 << ' ';          }          std::cout << '\n';      }  }  template <class T>  sparseMatrix<T> &sparseMatrix<T>::operator=(const sparseMatrix<T> &theMatrix)  {      terms.clear();      cols = theMatrix.cols;      rows = theMatrix.rows;      int \_size = theMatrix.terms.size();      int cnt = 0;  *while* (\_size-- and cnt < theMatrix.terms.size())      {          set(theMatrix.terms.get(cnt), cnt);          ++cnt;      }  *return* \*this;  }  int main()  {      int w = 0;      std::cin >> w;      sparseMatrix<int> P;  *while* (w--)      {          int ctrl;          std::cin >> ctrl;  *switch* (ctrl)          {  *case* 1:          {              int n, m;              std::cin >> n >> m;              P.reset(n, m);              int cnt = 0;  *for* (int i = 1; i <= n; ++i)              {  *for* (int j = 1; j <= m; ++j)                  {                      int temp;                      std::cin >> temp;  *if* (temp != 0)                      {                          matrixTerm<int> mt(i, j, temp);                          P.set(mt, cnt++);                      }                  }              }          }  *break*;  *case* 2:          {              int n, m, t;              std::cin >> n >> m >> t;              sparseMatrix<int> Q(n, m);              int cnt = 0;  *while* (t--)              {                  int x, y, v;                  std::cin >> x >> y >> v;                  matrixTerm<int> mt(x, y, v);                  Q.set(mt, cnt++);              }              P.multi(Q);          }  *break*;  *case* 3:          {              int n, m, t;              std::cin >> n >> m >> t;              sparseMatrix<int> Q(n, m);              int cnt = 0;  *while* (t--)              {                  int x, y, v;                  std::cin >> x >> y >> v;                  matrixTerm<int> mt(x, y, v);                  Q.set(mt, cnt++);              }              P.add(Q);          }  *break*;  *case* 4:          {              P.output();          }  *break*;  *case* 5:          {              P.transpose();          }  *break*;  *default*:  *break*;          }      }  } | | | |