山东大学 软件 学院

数据结构 课程实验报告

|  |  |  |  |
| --- | --- | --- | --- |
| 学号：201400301034 | 姓名： 石兴帮 | | 班级： 软件八班 |
| 实验题目：矩阵和三列表 | | | |
| 实验学时：4h | | 实验日期： 2015-12-13 | |
| 实验目的：  掌握特殊矩阵和稀疏矩阵。  掌握散列表及其应用。 | | | |
| 硬件环境：  MacBook Pro  OS X Yosemite 10.10.3 | | | |
| 软件环境：  Xcode 6.4 | | | |
| 实验内容与设计：   1. 实验内容（题目内容，输入要求，输出要求）   1.创建三对角矩阵类，采用按列映射方式，提供sotre和retrieve方法。  2.创建下三脚矩阵累，采用按列映射方式，提供store和retrieve方法。  3.创建稀疏矩阵类，采用行主顺序把稀疏矩阵映射到一维数组中，实现稀疏矩阵的转置和两个稀疏矩阵的加法操作。  4.使用散列表设计实现一个字典，假设关键字为整数且D为961，在字典中插入随机产生的500个不同的整数，实现字典的建立和搜索操作。分别使用线性开型寻址和链表散列解决溢出。  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）  1.三对角矩阵，即三条对角线上有值（低对角线，主对角线，高对角线）。  Store方法，根据j的值来判断就可以了。首先判断一下（i，j）位置应不应该有值，考虑一下他们差的绝对值，如果小于等于1的话就继续接下来的运算。显然，j＝1的时候，i的值就是它在数组中的位置。j不等于1的时候，首先加上第一列的2，然后在第j列之前的每一列都是3个元素，加上3\*（j－2），再考虑第j列本身。因为第j列有j－2个0，于是用行数i剪掉（j－2）就得到了第j列需要加上的值。综上，j不等于1时，它在数组中是2+3\*（j－2）＋i－j＋2，即2j＋i－2。在真正放到数组t中时，再－1即可。  Retrieve方法，根据映射公式从数组中提取就行。  2.下三角矩阵，  Store方法，在i》＝j的时候存储。先考虑第j列，显然第j列有j－1个0，那第j列要加上的就是i－（j－1）。再考虑前j－1列，第j－1列有n－j＋2个元素，第一列有n个元素，于是他们的总和是（2\*n－j＋2）＊（j－1）／2+i－j＋1。  Retrieve方法，根据映射公式从数组中提取。  3.稀疏矩阵转置：  对于一个普通的矩阵，我们转置它是很简单的，一个双重for循环，i，j反着来就转置好了。然而对一个稀疏矩阵而言，它的组成比普通矩阵复杂，每个位置都是一个Term对象，里面存放行，列以及值，一个稀疏矩阵可以由一个Term数组来表示。转置的第一个步骤是新建一个空的，同样大小但是行列相反的矩阵b。好，开始转置，试想我们直接从原矩阵的terms数组中提取元素，获取row，col，value，放到新矩阵b的terms数组中，但是这样产生的terms是没有顺序的。所以使用ColSize和RowNext两个数组，其中ColSize[i]是指矩阵第i列中的非0元素数，RowNext[i]代表转置矩阵第i行的下一个非0元素在b中的位置。    4.加法：随便加加就好了。  5.线性开型寻址：将数组建立成环形，遇到重复则填到它的下一个位置。  6.链表：建立D条链表，依次填值。  3.测试结果（测试输入，测试输出）    **1 3 0 0**  **2 4 6 0**  **0 5 7 9**  0 0 8 10  1 0 0 0  2 5 0 0  3 6 8 0  4 7 9 10  Enter number of rows, columns, and terms   3 4 3   Enter row, column, and value of term1   1 2 1   Enter row, column, and value of term2   2 3 2   Enter row, column, and value of term3   3 1 4   rows = 3columns = 4   nonzero terms = 3   a(1,2)=1   a(2,3)=2   a(3,1)=4   0 1 0 0   0 0 2 0   4 0 0 0   rows = 4columns = 3   nonzero terms = 3   a(1,3)=4   a(2,1)=1   a(3,2)=2   0 0 4   1 0 0   0 2 0   0 0 0   Enter number of rows, columns, and terms   4 3 4   Enter row, column, and value of term1   1 2 1   Enter row, column, and value of term2   2 1 3   Enter row, column, and value of term3   4 2 2   Enter row, column, and value of term4   3 1 2   rows = 4columns = 3   nonzero terms = 6   a(1,2)=1   a(1,3)=4   a(2,1)=4   a(3,2)=2   a(4,2)=2   a(3,1)=2   0 1 4   4 0 0   2 2 0   0 2 0  成功插入了： 500 个数  要查找的元素是：811, 它的位置是： 不存在  要查找的元素是：616, 它的位置是： 616  要查找的元素是：891, 它的位置是： 不存在  要查找的元素是：671, 它的位置是： 671  要查找的元素是：728, 它的位置是： 不存在  要查找的元素是：671, 它的位置是： 671  要查找的元素是：661, 它的位置是： 不存在  要查找的元素是：45, 它的位置是： 不存在  要查找的元素是：120, 它的位置是： 不存在  要查找的元素是：240, 它的位置是： 不存在    4.实现源代码（程序风格清晰易理解，有充分的注释）  //  //  shiyan4.h  //  c++初体验  //  //  Created by SXB on 15/11/19.  //  Copyright (c) 2015年 apple. All rights reserved.  //  #ifndef c\_\_\_\_\_\_shiyan4\_h  #define c\_\_\_\_\_\_shiyan4\_h  #include <iostream>  using namespace std;  //  MARK:  三对角矩阵  template<class T>  class TridiagonalMatrix {  public:      TridiagonalMatrix(int size =10)      {          n=size;          t=new T[3\*n-2];      }      ~TridiagonalMatrix() {delete []t;}      TridiagonalMatrix<T>& Store(const T&x, int i, int j);      T Retrieve(int i,int j) const;    private:      int n;  //存储维数      T \*t;   //存储三对角矩阵的一维数组  };  template<class T>  TridiagonalMatrix<T>& TridiagonalMatrix<T>::Store(const T& x, int i, int j)  {  //    把x存为T(i,j)      if (i<1 || j<1 || i>n || j>n ) {          //  throw OutOfBounds();      }      if (i-j<-1 || i-j>1) {          //  throw MustBeZero();      }      switch (j) {          case 1:              t[i-1] = x;              break;            default:              t[2\*j+i-3] = x;              break;      }      return \*this;  }  template<class T>  T TridiagonalMatrix<T>::Retrieve(int i, int j) const  {  //    返回T(i,j)      if (i<1 || j<1 || i>n || j>n ) {          //  throw OutOfBounds();      }      if (i-j<-1 || i-j>1) {          return 0;      }      switch (j) {          case 1:              return t[i-1];            default:              return t[2\*j+i-3];      }  }  //  MARK:  下三角矩阵  template<class T>  class LowerMatrix{  public:      LowerMatrix(int size =10)      {          n=size;          t = new T[n\*(n+1)/2];      }      ~LowerMatrix() {delete []t;}      LowerMatrix<T>& Store(const T& x, int i, int j);      T Retrieve(int i, int j) const;    private:      int n;//   矩阵维数      T \*t; //   存储下三角矩阵的一维数组  };  template<class T>  LowerMatrix<T>& LowerMatrix<T>:: Store(const T& x, int i , int j)  {  //    把x存为L(i,j)      if (i<1 || j<1 || i>n || j>n ) {          //  throw OutOfBounds();      }  //    当且仅当i>=j时(i,j)位于下三角      if (i>=j) {          t[(2\*n-j+2)\*(j-1)/2+i-j+1] = x;      }      else if(x!=0){  //        throw MustBeZero()      }      return \*this;  }  template<class T>  T LowerMatrix<T>::Retrieve(int i, int j) const  {  //    返回 L(i,j).      if (i<1 || j<1 || i>n || j>n ) {          //  throw OutOfBounds();      }      //    当且仅当i>=j时(i,j)位于下三角      if (i>=j) {          return t[(2\*n-j+2)\*(j-1)/2+i-j+1];      }else {          return 0;      }  }  //  MARK:  稀疏矩阵  template<class T>  struct Term {      int row,col;      T value;  };  template<class T>  class SparseMatrix  {  public:      SparseMatrix(int maxTerms = 10);      ~SparseMatrix() {delete [] a;}      void Transpose(SparseMatrix<T> &b) const;      void Add(const SparseMatrix<T> &b, SparseMatrix<T> &c) const;      void shuru();      void shuchu();    //private:      void Append(const Term<T> &t);      int rows,cols;  //矩阵维数      int terms;  //非0元素个数      Term<T> \*a; //存储非0元素的数组      int MaxTerms;   //数组a的大小  };  template<class T>  SparseMatrix<T>::SparseMatrix(int maxTerms)  {  //    稀疏矩阵的构造函数      if (maxTerms<1) {          //throw BadInitializers();      }      MaxTerms = maxTerms;      a = new Term<T>[MaxTerms];      terms = rows = cols = 0;  }  template<class T>  void SparseMatrix<T>::shuchu()  {      //    把\*this 送至输出流      //    输出矩阵的特征      cout << "rows = " << rows << "columns = " << cols << endl;      cout << "nonzero terms = " << terms << endl;      //    输出非0元素，每行1个      int shuchutemp[rows][cols];      for (int i=0; i<rows; i++) {          for (int j=0; j<cols; j++) {              shuchutemp[i][j]=0;          }      }      for (int i = 0; i < terms; i++) {          cout << "a(" << a[i].row << "," << a[i].col << ")=" << a[i].value << endl;          shuchutemp[a[i].row-1][a[i].col-1] = a[i].value;      }      for (int i=0; i<rows; i++) {          for (int j=0; j<cols; j++) {              cout << shuchutemp[i][j] << " ";          }          cout << endl;      }  }  template<class T>  void SparseMatrix<T>::shuru()  {      //    输入一个稀疏矩阵      //    输入矩阵的特征      cout << "Enter number of rows, columns, and terms" << endl;      cin >> rows >> cols >> terms;      if (terms > MaxTerms) {          //        throw NoMem();      }  //        输入矩阵元素      for (int i=0; i<terms; i++) {          cout << "Enter row, column, and value of term" << (i+1) << endl;          cin >> a[i].row >> a[i].col >> a[i].value;      }  }  template<class T>  void SparseMatrix<T>::Transpose(SparseMatrix<T> &b) const  {  //  把\*this 的转置结果送入 b  //    确信b 有足够的空间  //    if (terms > b.Maxterms) {  //        throw NoMem();  //    }  //    设置转置特征      b.cols = rows;      b.rows = cols;      b.terms = terms;  //    初始化      int \*ColSize, \*RowNext;      ColSize = new int[cols +1];      RowNext = new int[rows + 1];  //    计算 \*this 每一列的非0元素数      for (int i=1; i <= cols ; i++) {          ColSize[i] = 0;      }      for (int i=0; i<terms; i++) {          ColSize[a[i].col]++;      }  //    给出b 中每一行的起始点      RowNext[1] = 0;      for (int i=2; i <= cols; i++) {          RowNext[i] = RowNext[i-1] + ColSize[i-1];      }  //    进行转置操作      for (int i=0; i < terms; i++) {          int j= RowNext[a[i].col]++; //  在b 中的位置          b.a[j].row = a[i].col;          b.a[j].col = a[i].row;          b.a[j].value = a[i].value;      }  }  template<class T>  void SparseMatrix<T>::Append(const Term<T> &t)  {  //    把一个非0元素 t添加到 \*this之中      if (terms >= MaxTerms) {  //        throw NoMem();      }      a[terms] = t;      terms++;  }  template<class T>  void SparseMatrix<T>::Add(const SparseMatrix<T> &b, SparseMatrix<T> &c ) const  {  //    计算 c＝ (\*this) + b  //    验证可行性      if (rows!= b.rows || cols != b.cols) {  //        throw SizeMisMatch()      }  //    设置结果矩阵c的特征      c.rows = rows;      c.cols = cols;      c.terms = 0; // 初值  //    定义 \*this 和 b的游标      int ct=0, cb =0;  //    在\*this 和 b中遍历      while (ct < terms && cb< b.terms) {  //        每一个元素的行主索引          int indt = a[ct].row \* cols + a[ct].col;          int indb = b.a[cb].row \* cols + b.a[cb].col;          if (indt < indb) {  //            b的元素在后              c.Append(a[ct]);              ct++; // \*this 的下一个元素          }          else {              if (indt == indb) {  //                位置相同  //                仅当和不为0时才添加到 c中                  if (a[ct].value + b.a[cb].value) {                      Term<T> t;                      t.row = a[ct].row;                      t.col = a[ct].col;                      t.value = a[ct].value + b.a[cb].value;                      c.Append(t);                  }                  ct++;cb++;              }              else              {                  c.Append(b.a[cb]);                  cb++; // b的下一个元素              }          }      }  //    复制剩余元素      for (; ct<terms; ct++) {          c.Append(a[ct]);      }      for (; cb < b.terms; cb++) {          c.Append(b.a[cb]);      }  }  //  MARK:  散列表  template<class E,class K>  class HashTable{  public:      HashTable(int divisor = 11);      ~HashTable() {delete [] ht; delete [] empty;}      bool Search(const K& k, E& e) const;      HashTable<E,K>& Insert(const E& e);    //private:      int hSearch(const K& k) const;      int D; //  散列函数的除数      E \*ht; //  散列数组      bool \*empty; //  一维数组  };  template<class E, class K>  HashTable<E, K>::HashTable(int divisor)  {  //    构造函数      D = divisor;    //    分配散列数组      ht = new E[D];      empty = new bool[D];    //    将所有桶置空      for (int i=0; i<D ; i++) {          empty[i] = true;      }  }  template<class E,class K>  int HashTable<E,K>::hSearch(const K& k) const  {  //    查找一个开地址表  //    如果存在，则返回k的位置  //    否则返回插入点（如果有足够空间）      int i = k%D; //  起始桶      int j = i; //  在起始桶处开始      do {          if (empty[j] || ht[j] == k) {              return j;          }          j = (j+1) % D; //  下一个桶      }      while (j != i ); //  又返回起始桶？      return j; //  表已经满  }  template<class E, class K>  bool HashTable<E,K>::Search(const K& k, E &e) const  {  //    搜索与k匹配的元素并放入e  //    如果不存在这样的元素，则返回false      int b = hSearch(k);      if (empty[b] || ht[b] != k) {          return false;      }      e = ht[b];      return true;  }  template<class E, class K>  HashTable<E, K>& HashTable<E, K>::Insert(const E& e)  {  //  在散列表中插入      K k=e; //  抽取key的值。当E为用户自定义的类或数据类型时，有必要重载%, !=, ==等操作符      int b = hSearch(k);    //    检查是否能完成插入      if (empty[b]) {          empty[b] = false;          ht[b] = e;          return \*this;      }    //    不能插入，检查是否有重复值或表满      if (ht[b] == k) {          int count =0;          while (count < D) {              b = (b+1)%D;              count ++;              if (empty[b]) {                  empty[b] = false;                  ht[b] = e;                  return \*this;              }          }  //        throw BadInput();      }else {  //        throw NoMem();      }      return \*this;  }  // MARK:  链表散列  template<class E, class K>  struct SortedChainNode {      K key;      E element;      SortedChainNode<E, K> \*link;  };  template<class E, class K>  class SortedChain{  public:      SortedChain() {first = 0;}  //    ~SortedChain();      bool IsEmpty() const {return first == 0;}      int Length() const;      bool Search(const K& k, E& e) const;      SortedChain<E, K>& Delete(const K& k, E& e);      SortedChain<E, K>& Insert(const E& e);        SortedChainNode<E, K> \*first;  };  template<class E, class K>  bool SortedChain<E, K>::Search(const K& k, E &e) const  {  //    搜索与k匹配的元素，结果放入e  //    如果没有匹配的元素，则返回false      SortedChainNode<E, K> \*p =first;    //    搜索与k相匹配的元素      for (; p && p->element < k; p = p->link);    //    验证是否与k匹配      if (p && p->element == k) {          e = p->element;          return true;      }      return false;  }  template<class E, class K>  SortedChain<E, K>& SortedChain<E, K>::Insert(const E& e)  {      SortedChainNode<E, K> \*p = first, \*tp = 0; //  跟踪p    //    移动tp以便把e插入到tp之后      for (; p && p->element < e; tp = p, p = p->link);        SortedChainNode<E, K> \*q = new SortedChainNode<E, K>;      q->element = e;        q->link = p;      if (tp) {          tp->link = q;      }else          first = q;        return \*this;  }  template<class E, class K>  class ChainHashTable{  public:      ChainHashTable(int divisor = 10)      {          D = divisor;          ht = new SortedChain<E, K>[D];      }      ~ChainHashTable() {delete [] ht;}      bool Search(const K& k, E& e) const      {          return ht[k%D].Search(k,e);      }      ChainHashTable<E, K>& Insert(const E& e)      {          ht[e%D].Insert(e);          return \*this;      }      void Output() const {          for (int i=0; i<D; i++) {              cout << "第" << i+1 << "条链表：";              SortedChain<E, K> tmp = ht[i];              SortedChainNode<E, K> \*p = tmp.first;              for (; p && p->element; p = p->link) {                  cout << p->element << " ";              }              cout << endl;          }      }  //private:      int D;      SortedChain<E, K> \*ht;  };  #endif  //#include "shiyan4.h"  //#include <iostream>  //#include<stdlib.h>  //#define random(x) (rand()%x)  //  //using namespace std;  //  //int main(){  ////    MARK：  三对角  ////    TridiagonalMatrix<int> \*test = new TridiagonalMatrix<int>(4);  ////    test->Store(1,1,1);  ////    test->Store(2, 2, 1);  ////    test->Store(3, 1, 2);  ////    test->Store(4, 2, 2);  ////    test->Store(5, 3, 2);  ////    test->Store(6, 2, 3);  ////    test->Store(7, 3, 3);  ////    test->Store(8, 4, 3);  ////    test->Store(9, 3, 4);  ////    test->Store(10, 4, 4);  ////    for (int i=1; i<=4; i++) {  ////        for (int j=1; j<=4; j++) {  ////            cout << test->Retrieve(i, j) << " ";  ////        }  ////        cout << endl;  ////    }  ////    MARK：  下三角  ////    LowerMatrix<int> \*test = new LowerMatrix<int>(4);  ////    test->Store(1, 1, 1);  ////    test->Store(2, 2, 1);  ////    test->Store(3, 3, 1);  ////    test->Store(4, 4, 1);  ////    test->Store(5, 2, 2);  ////    test->Store(6, 3, 2);  ////    test->Store(7, 4, 2);  ////    test->Store(8, 3, 3);  ////    test->Store(9, 4, 3);  ////    test->Store(10, 4, 4);  ////  ////    for (int i=1; i<=4; i++) {  ////        for (int j=1; j<=4; j++) {  ////            cout << test->Retrieve(i, j) << " ";  ////        }  ////        cout << endl;  ////    }  //  ////    MARK：  稀疏矩阵操作  //  ////    SparseMatrix<int> \*test = new SparseMatrix<int>(20);  ////    test->shuru();  ////  ////    test->shuchu();  ////    SparseMatrix<int> \*temp = new SparseMatrix<int>(20);  ////    test->Transpose(\*temp);  ////    temp->shuchu();  ////  ////    SparseMatrix<int> \*temp2 = new SparseMatrix<int>(20);  ////    temp2->shuru();  ////  ////    SparseMatrix<int> \*temp3 = new SparseMatrix<int>(20);  ////    temp->Add(\*temp2, \*temp3);  ////    temp3->shuchu();  //  //  ///\*  稀疏矩阵的一组测试如下：  // Enter number of rows, columns, and terms  // 3 4 3  // Enter row, column, and value of term1  // 1 2 1  // Enter row, column, and value of term2  // 2 3 2  // Enter row, column, and value of term3  // 3 1 4  // rows = 3columns = 4  // nonzero terms = 3  // a(1,2)=1  // a(2,3)=2  // a(3,1)=4  // 0 1 0 0  // 0 0 2 0  // 4 0 0 0  // rows = 4columns = 3  // nonzero terms = 3  // a(1,3)=4  // a(2,1)=1  // a(3,2)=2  // 0 0 4  // 1 0 0  // 0 2 0  // 0 0 0  // Enter number of rows, columns, and terms  // 4 3 4  // Enter row, column, and value of term1  // 1 2 1  // Enter row, column, and value of term2  // 2 1 3  // Enter row, column, and value of term3  // 4 2 2  // Enter row, column, and value of term4  // 3 1 2  // rows = 4columns = 3  // nonzero terms = 6  // a(1,2)=1  // a(1,3)=4  // a(2,1)=4  // a(3,2)=2  // a(4,2)=2  // a(3,1)=2  // 0 1 4  // 4 0 0  // 2 2 0  // 0 2 0  //  //\*/  ////  MARK:  线性开型寻址  ////    HashTable<int, int> \*test = new HashTable<int, int>(961);  ////  ////    for (int i=0; i<500; i++) {  ////        int x = random(1000);  ////        test->Insert(x);  ////    }  ////    int count=0;  ////    for (int j=0; j<961; j++) {  ////        if (!test->empty[j]) {  ////            count++;  ////        }  ////    }  ////    cout << "成功插入了： "<< count <<" 个数" << endl;  ////  ////    for (int k=0; k<10; k++) {  ////        int x = random(1000);  ////        cout << "要查找的元素是：" << x << ", 它的位置是： ";  ////        int y;  ////        if (test->Search(x, y)) {  ////            cout << test->hSearch(x) << endl;  ////        }else  ////            cout << "不存在" << endl;  ////    }  //  ////    MARK:  链表散列  //    ChainHashTable<int, int> \*test = new ChainHashTable<int, int>(961);  //  //    for (int i=0; i<500; i++) {  //        int x= random(1000);  //        test->Insert(x);  //    }  //    test->Output();  //  //    for (int j=0; j<10; j++) {  //        int x = random(1000);  //        int y;  //        if (test->Search(x, y)) {  //            cout << x << "的位置是第：" << y%961 << "条链表" << endl;  //        }else  //            cout << "链表中没有" << x << endl;  //    }  //} | | | |
| 结论分析与体会：  学到了很多知识。 | | | |