中北大学大数据学院

实验报告

课 程 名 称 算 法 分 析 与 设 计 学 号 1807004528学 生 姓 名 蔡帅教 师 井 超

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **专业** | **数据科学与**  **大数据技术** | **实验室名称** |  | **实验时间** | 2020年3月7日 |
| **实验名称**  程序一：用LU矩阵分解求解逆矩阵问题。  程序二：AVL平衡树 | | | | | |
| **实验目的**  程序一：学习LU分解求解逆矩阵对于程序计算速度方面的提高。  程序二：学习平衡平衡树的建立 | | | | | |
| **实验题目**  程序一：使用c语言，用二维数组模拟矩阵LU分解并求逆矩阵的过程。  程序二：建立AVL平衡树 | | | | | |
| **实验源代码**  **1.**  **import** java.util.Arrays; **import** java.util.List;  **public class** Main{   **public static void** main(String[] args) {  **double** [][]A ={{2,4,-2},{4,9,-3},{-2,-1,7}};  **double** []b = {2,8,10};  **int** row = 3;  **double**[]x = solve(A, b);  **for**(**int** i = 0;i<x.length;i++){  System.out.println(x[i]);  }   }   **private static double**[] solve(**double**[][] a, **double**[] b) {  List<**double**[][]> LAndU = decomposition(a); *//LU decomposition* **double**[][] L = LAndU.get(0);  **double**[][] U = LAndU.get(1);  **double**[] UMultiX = getUMultiX(a, b, L); *//前代* **return** getSolution(a, U, UMultiX); *//回代* }   */\*\*  \* Get solution of the equations   \** ***@param*** *a - Coefficient matrix of the equations   \** ***@param*** *U - U of LU Decomposition   \** ***@param*** *UMultiX - U multiply X   \** ***@return*** *Equations solution   \*/* **private static double**[] getSolution(**double**[][] a, **double**[][] U,  **double**[] UMultiX) {  **double**[] solutions = **new double**[a[0].length];  **for**(**int** i=U.length-1; i>=0; i--) {  **double** right\_hand = UMultiX[i];  **for**(**int** j=U.length-1; j>i; j--) {  right\_hand -= U[i][j] \* solutions[j];  }  solutions[i] = right\_hand / U[i][i];  }  **return** solutions;  }   */\*\*  \* Get U multiply X   \** ***@param*** *a - Coefficient matrix of the equations   \** ***@param*** *b - right-hand side of the equations   \** ***@param*** *L - L of LU Decomposition   \** ***@return*** *U multiply X   \*/* **private static double**[] getUMultiX(**double**[][] a, **double**[] b, **double**[][] L) {  **double**[] UMultiX = **new double**[a.length];  **for**(**int** i=0; i<a.length; i++) {  **double** right\_hand = b[i];  **for**(**int** j=0; j<i; j++) {  right\_hand -= L[i][j] \* UMultiX[j]; *//* }  UMultiX[i] = right\_hand / L[i][i];  }  **return** UMultiX;  }   **private static** List<**double**[][]> decomposition(**double**[][]a){  **double**[][] L = createIndentityMatrix(a.length);   **for**(**int** j=0; j<a[0].length - 1; j++) {  **if**(a[j][j] == 0) {  **throw new** IllegalArgumentException(**"zero pivot encountered."**);  }   **for**(**int** i=j+1; i<a.length; i++) {  **double** mult = a[i][j] / a[j][j];  **for**(**int** k=j; k<a[i].length; k++) {  a[i][k] = a[i][k] - a[j][k] \* mult;  *//得出上三角矩阵U,通过减去矩阵的第一行,第二行,第一行(第二行)得到上三角矩阵* }  L[i][j] = mult; *//得到下三角矩阵是得出上三角矩阵的乘积因子* }  }  **return** Arrays.asList(L, a);   }  **private static double**[][]createIndentityMatrix(**int** row){  **double**[][]identityMatrix = **new double**[row][row];  **for**(**int** i=0;i<identityMatrix.length;i++){  **for**(**int** j=i;j<identityMatrix[i].length;j++){  **if**(j == i){  identityMatrix[i][j]= 1;  }  }  }  **return** identityMatrix;  } }  2.**import** java.nio.BufferUnderflowException; **import** java.util.Comparator;  */\*\*  \* 为二叉查找树添加了  \* 高度，平衡判断，单双旋  \*  \* 可以通过 T extends Comparable 要求 T 必须实现该接口  \*/* **public class** AVLTree<T> {   **private** AVLNode<T> root;  **private** Comparator<? **super** T> cmp;   **public** AVLTree(){  **this**.root = **null**;  }  *//传入比较器* **public** AVLTree(Comparator<? **super** T> c){  **this**.root = **null**;  **this**.cmp = c;  }   *//通过比较器比较（不需要T实现Comparable接口）* **private int** myCompare(T lhs, T rhs){  **if** (cmp != **null**){  **return** cmp.compare(lhs, rhs);  }**else** {  **return** ((Comparable)lhs).compareTo(rhs);  }  }   **public void** makeEmpty(){  root = **null**;  }  **public boolean** isEmpty(){  **return** root == **null**;  }   **public boolean** contains(T x){  **return** contains(x, root);  }  **public** T findMin(){  **if** (isEmpty()){  **throw new** BufferUnderflowException();  }  **return** findMin(root).element;  }  **public** T findMax(){  **if** (isEmpty()){  **throw new** BufferUnderflowException();  }  **return** findMax(root).element;  }   **public void** insert(T x){  root = insert(x, root);  }  **public void** remove(T x){  root = remove(x, root);  }  **public void** printTree(){  **if** (isEmpty()){  System.out.println(**"Empty tree"**);  }**else** {  printTree(root);  }  }   */\*\*  \* private 方法  \*/* **private boolean** contains(T x, AVLNode<T> t){  **if** (t == **null**){  **return false**;  }   **int** compareResult = myCompare(x, t.element);   **if** (compareResult < 0){  **return** contains(x, t.left);  }**else if** (compareResult > 0){  **return** contains(x, t.right);  }**else** {  **return true**;  }  }  *//递归* **private** AVLNode<T> findMin(AVLNode<T> t){  **if** (t == **null**){  **return null**;  }**else if** (t.left == **null**){  **return** t;  }  **return** findMin(t.left);  }  *//非递归* **private** AVLNode<T> findMax(AVLNode<T> t){  *//注意不要改变t的引用* **if** (t != **null**){  **while**(t.right != **null**){  t = t.right;  }  }  **return** t;  }   */\*\*  \* 插入  \* 找到要插入元素应该在的位置  \* 新建一个包含x的Node赋值给对应指针  \*/* **private** AVLNode<T> insert(T x, AVLNode<T> t){  *//递归出口（新建一个包含x的Node赋值给对应的t.left/right）* **if** (t == **null**){  **return new** AVLNode<>(x, **null**, **null**);  }   **int** compareResult = myCompare(x, t.element);   **if** (compareResult < 0){  t.left = insert(x, t.left);  }**else if** (compareResult > 0){  t.right = insert(x, t.right);  }  *//如果重复，可以用附加域表示频率* **return** balance(t);  }  */\*\*  \* 删除  \* 查找元素，找到后使用目标元素右子树的最小结点代替  \* 并删除目标元素右子树的最小节点  \* 效率不高（可以采用懒惰删除）  \*/* **private** AVLNode<T> remove(T x, AVLNode<T> t){  *//直到t == null，切断连接* **if** (t == **null**){  **return** t;  }   **int** compareResult = myCompare(x, t.element);   **if** (compareResult < 0){  t.left = remove(x, t.left);  }**else if** (compareResult > 0){  t.right = remove(x, t.right);  }  *//找到要被删除的元素，它的左右两个节点都 != null* **else if** (t.left != **null** && t.right != **null**){  *//以目标右子树的最小数据代替（该数据总比目标大）  // 然后递归删除右子树最小的节点* t.element = findMin(t.right).element;  t.right = remove(t.element, t.right);  }**else** {  */\*\*  \* 如果目标只有一个 != null 的节点  \* 直接走向 != null 节点，并与目标节点的上一个节点连接  \* 如果没有左右节点，则直接走向右边，即 t = null;  \*/* t = (t.left != **null**) ? t.left : t.right;  }  **return** balance(t);  }    *//平衡判断常量* **private static final int** ALLOWED\_IMBALANCE = 1;  *//平衡判断，单双旋转（旋转后高度会 + 1 / 唯一改变高度的方法）* **private** AVLNode<T> balance(AVLNode<T> t){  **if** (t == **null**){  **return** t;  }   *//判断左右旋的四种情况* **if** (height(t.left) - height(t.right) > ALLOWED\_IMBALANCE){  *//使用 >= 是为了保证删除时两子树一样高时使用单旋* **if** (height(t.left.left) >= height(t.left.right)){  *//保证根节点的连接* t = rotateWithLeftChild(t);  }**else** {  t = doubleWithLeftChild(t);  }  }**else if** (height(t.right) - height(t.left) > ALLOWED\_IMBALANCE) {  **if** (height(t.right.right) >= height(t.right.left)) {  *//外侧单旋* t = rotateWithRightChild(t);  }  *//内侧双旋* **else** {  t = doubleWithRightChild(t);  }  }   *//更新高度* t.height = Math.max(height(t.left), height(t.right)) + 1;  **return** t;  }  *//快速获取高度* **private int** height(AVLNode<T> t){  **return** t == **null** ? -1 : t.height;  }  *//单旋转（左边高）（旋转传入都是根节点）* **private** AVLNode<T> rotateWithLeftChild(AVLNode<T> k2){  *//找到 k1* AVLNode<T> k1 = k2.left;  *//k1 的子树移动* k2.left = k1.right;  *//根节点转换* k1.right = k2;  *//高度更新* k2.height = Math.max(height(k2.left), height(k2.right)) + 1;  k1.height = Math.max(height(k1.left), k2.height) + 1;  *//返回根节点* **return** k1;  }  *//单旋转（右边高）* **private** AVLNode<T> rotateWithRightChild(AVLNode<T> k2){  *//找到 k1* AVLNode<T> k1 = k2.**right**;  *//k1 的子树移动* k2.**right** = k1.**left**;  *//根节点转换* k1.**left** = k2;  *//高度更新* k2.**height** = Math.*max*(height(k2.**right**), height(k2.**left**)) + 1;  k1.**height** = Math.*max*(height(k1.**right**), k2.**height**) + 1;  *//返回根节点* **return** k1;  }  *//右 - 左 双旋转（左边高）* **private** AVLNode<T> doubleWithLeftChild(AVLNode<T> k3){  *//k3.left 右旋转，k3 左旋转* k3.**left** = rotateWithRightChild(k3.**left**);  **return** rotateWithLeftChild(k3);  }  *//左 - 右 双旋转（右边高）* **private** AVLNode<T> doubleWithRightChild(AVLNode<T> k3){  k3.**right** = rotateWithLeftChild(k3.**right**);  **return** rotateWithRightChild(k3);  }   *//中序遍历* **private void** printTree(AVLNode<T> t){  **if**(t != **null**){  printTree(t.**left**);  System.out.print(t.element + **"\t"**);  printTree(t.right);  }  }   *//练习！！！  //判断二叉树是否平衡* **public boolean** isBalance(){  **return** checkBalance(root);  }  **private boolean** checkBalance(AVLNode<T> root) {  *//空树平衡* **if**(root == **null**)  **return true**;  *//求左右节点深度（左右子树高度）* **int** right = deep(root.right);  **int** left = deep(root.left);  *//不平衡* **if**(Math.abs(left - right) > 1)  **return false**;  *//左右子树都平衡* **return** checkBalance(root.right) && checkBalance(root.left);  }  *//递归求节点的深度* **private int** deep(AVLNode<T> t) {  **if**(t == **null**)  **return** 0;  **int** deepRight = deep(t.right);  **int** deepLeft = deep(t.left);  **return** 1 + (deepRight>deepLeft ? deepRight : deepLeft);  }    *//节点类（嵌套类）* **private static class** AVLNode<T>{  **public** T element;  **public** AVLNode<T> left;  **public** AVLNode<T> right;  *//储存该节点的高度* **public int** height;   **public** AVLNode(T ele){  **this**(ele, **null**, **null**);  }  **public** AVLNode(T ele, AVLNode<T> lt, AVLNode<T> rt){  **this**.element = ele;  **this**.left = lt;  **this**.right = rt;  height = 0;  }  } }  **public class** Main {   **public static void** main(String[] args) {   AVLTree<Integer> tree = **new** AVLTree<>(Integer::*compare*);   **for** (**int** i=0;i<10;i++){  tree.insert(i);  }  tree.printTree();  System.out.println();  System.out.println(tree.isBalance());  } } | | | | | |
| **实验结果**（给出题目的运行结果（插入运行结果截图））  1.  IMG_256  2.  IMG_256 | | | | | |

格式要求：各单元格空白行尽量少。

提交要求：电子版，文件命名中将学号和姓名具体化。