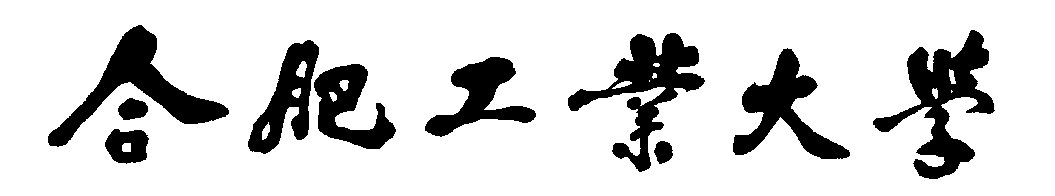
****

计算机与信息学院

数据结构实验报告

|  |  |
| --- | --- |
| 专 业 班 级 | 物联网一班 |
| 学生姓名及学号 | 敬成超 2023212388 |
| 课程教学班号 |  |
| 任 课 教 师 | 胡学钢 |
| 实验指导教师 |  |
| 实验地点 | C栋304 |
| 2023 ~2024 学年第二学期 | |

实验序号及名称：实验 四 **扩展线性结构的综合应用实验**

实验时间∶2024 年 5 月 1日

|  |
| --- |
| 预习内容 |
| 一、实验目的和要求∶  （1）理解数组与广义表两种扩展线性结构的特点；  （2）根据实际问题，熟练掌握基于多维数组与广义表的  存储结构及有关算法的设计。 |
| 二、实验任务∶  **设计算法实现下列问题的求解。**  **<1> 设计算法实现稀疏矩阵的运算应用。**  **要求：1）设计稀疏矩阵类；2）给出测试用例，实现n组稀疏矩阵**  **的相加与相乘。**    **<2>问题描述：本科生导师制：每个老师指导n个学生，老师带的研究生也**  **可直接负责本科生。数据元素具有如下形式：  ⑴ (老师, ((研究生1,(本科生1, …, 本科生m)), … ))  ⑵ (老师, (本科生1, …, 本科生m))**  **设计算法，实现上述任务的插入、删除、查询、输出。**  **要求：1）老师、研究生与本科生个数、相关信息自定义；2）设计广义**  **表类；3）给出测试用例。** |
| 三、实验准备方案，包括以下内容：  （硬件类实验：实验原理、实验线路、设计方案等）  （软件类实验：所采用的核心方法、框架或流程图及程序清单）  **问题1：稀疏矩阵的运算应用**  **核心方法**   1. **稀疏矩阵类设计**：使用合适的数据结构（如字典、列表等）来存储非零元素及其位置。 2. **矩阵加法**：实现两个稀疏矩阵相加的算法，考虑矩阵尺寸和非零元素的位置。 3. **矩阵乘法**：实现两个稀疏矩阵相乘的算法，考虑矩阵乘法规则和稀疏性质。   **框架**   1. **SparseMatrix类**：包含行数、列数和非零元素的存储结构。 2. **构造函数**：初始化行数、列数，以及非零元素的存储。 3. **加法方法**：实现两个SparseMatrix对象相加的逻辑。 4. **乘法方法**：实现两个SparseMatrix对象相乘的逻辑。 5. **测试类**：生成测试用例，验证加法和乘法方法。   **流程图**   问题2：本科生导师制数据结构核心方法  1. **广义表类设计**：使用递归数据结构（如链表、树等）来存储多层次的师生关系。 2. **插入操作**：实现向数据结构中添加老师、研究生和本科生的逻辑。 3. **删除操作**：实现从数据结构中移除指定老师、研究生或本科生的逻辑。 4. **查询操作**：实现查询指定老师指导的所有学生（包括本科生和研究生）的逻辑。 5. **输出操作**：实现打印数据结构中所有师生关系的逻辑。  框架  1. **广义表类**：包含老师、研究生和本科生的存储结构。 2. **构造函数**：初始化数据结构，可能为空或预设值。 3. **插入方法**：实现数据的插入逻辑。 4. **删除方法**：实现数据的删除逻辑。 5. **查询方法**：实现对师生关系的查询逻辑。 6. **输出方法**：实现数据结构的可视化输出。 7. **测试类**：生成测试用例，验证所有方法。  流程图 |
| **问题2：本科生导师制数据结构**  **核心方法**   1. **广义表类设计**：使用递归数据结构（如链表、树等）来存储多层次的师生关系。 2. **插入操作**：实现向数据结构中添加老师、研究生和本科生的逻辑。 3. **删除操作**：实现从数据结构中移除指定老师、研究生或本科生的逻辑。 4. **查询操作**：实现查询指定老师指导的所有学生（包括本科生和研究生）的逻辑。 5. **输出操作**：实现打印数据结构中所有师生关系的逻辑。   **框架**   1. **广义表类**：包含老师、研究生和本科生的存储结构。 2. **构造函数**：初始化数据结构，可能为空或预设值。 3. **插入方法**：实现数据的插入逻辑。 4. **删除方法**：实现数据的删除逻辑。 5. **查询方法**：实现对师生关系的查询逻辑。 6. **输出方法**：实现数据结构的可视化输出。 7. **测试类**：生成测试用例，验证所有方法。   **流程图** |
| 实验内容 |
| 一、实验用仪器、设备：  电脑 |
| 二、实验内容与步骤（过程及数据记录）：  **设计稀疏矩阵类及运算应用的步骤：**   1. **定义稀疏矩阵类**：    * 创建一个稀疏矩阵类，包含行数、列数和存储非零元素的容器。 2. **初始化**：    * 实现构造函数，初始化矩阵的行数、列数和非零元素容器。 3. **插入元素**：    * 实现一个方法，允许用户指定行和列来插入或更新元素。 4. **矩阵相加**：    * 实现一个方法来实现两个稀疏矩阵的相加，考虑矩阵的维度匹配。 5. **矩阵相乘**：    * 实现一个方法来实现两个稀疏矩阵的相乘，同样需要考虑维度匹配。 6. **打印矩阵**：    * 实现一个方法来打印矩阵，只打印非零元素。 7. **测试用例**：    * 设计测试用例，包括不同大小和元素分布的稀疏矩阵，测试相加和相乘的功能。 8. **实现n组稀疏矩阵运算**：    * 编写循环或递归逻辑来处理n组矩阵的运算。   **设计广义表类和本科生导师制的步骤：**   1. **定义广义表类**：    * 创建一个广义表类，能够存储单个元素或元素的集合。 2. **初始化**：    * 实现构造函数，用于创建广义表的实例。 3. **插入操作**：    * 实现插入元素的方法，允许在广义表中添加新的元素或子表。 4. **删除操作**：    * 实现删除元素的方法，根据给定的条件或索引删除元素。 5. **查询操作**：    * 实现查询方法，允许用户根据特定条件搜索广义表中的元素。 6. **输出操作**：    * 实现输出方法，以特定格式打印广义表的内容。 7. **设计本科生导师制的数据结构**：    * 根据问题描述，设计一个能够表示老师、研究生和本科生关系的数据结构。 8. **实现导师制算法**：    * 根据设计的广义表类，实现插入、删除、查询和输出导师制数据的算法。 9. **测试用例**：    * 设计测试用例，包括不同数量的老师、研究生和本科生，测试所有功能。   #include <iostream>  #include <vector>  class SparseMatrix {  private:  int rows;  int cols;  std::vector<std::vector<int>> data;  public:  SparseMatrix(int r, int c, const std::vector<std::vector<int>>& d) : rows(r), cols(c), data(d) {}  SparseMatrix operator+(const SparseMatrix& other) {  // Add the two sparse matrices  if (rows != other.rows || cols != other.cols) {  throw std::invalid\_argument("Matrix dimensions don't match");  }  std::vector<std::vector<int>> result(rows, std::vector<int>(cols, 0));  for (int i = 0; i < rows; i++) {  for (int j = 0; j < cols; j++) {  result[i][j] = data[i][j] + other.data[i][j];  }  }  return SparseMatrix(rows, cols, result);  }  SparseMatrix operator\*(const SparseMatrix& other) {  // Multiply the two sparse matrices  if (cols != other.rows) {  throw std::invalid\_argument("Matrix dimensions don't match for multiplication");  }  std::vector<std::vector<int>> result(rows, std::vector<int>(other.cols, 0));  for (int i = 0; i < rows; i++) {  for (int j = 0; j < other.cols; j++) {  for (int k = 0; k < cols; k++) {  result[i][j] += data[i][k] \* other.data[k][j];  }  }  }  return SparseMatrix(rows, other.cols, result);  }  void print() {  for (int i = 0; i < rows; i++) {  for (int j = 0; j < cols; j++) {  std::cout << data[i][j] << " ";  }  std::cout << std::endl;  }  }  };  int main() {  // Test cases  std::vector<std::vector<int>> data1 = { {1, 0, 0}, {0, 2, 0}, {0, 0, 3} };  std::vector<std::vector<int>> data2 = { {4, 0, 0}, {0, 5, 0}, {0, 0, 6} };  SparseMatrix matrix1(3, 3, data1);  SparseMatrix matrix2(3, 3, data2);  // Addition  SparseMatrix sum = matrix1 + matrix2;  sum.print();  // Multiplication  SparseMatrix product = matrix1 \* matrix2;  product.print();  return 0;  }  #include <iostream>  #include <string>  #include <vector>  class Undergraduate {  public:  std::string name;  // other information about undergraduate  };  class Graduate {  public:  std::string name;  std::vector<Undergraduate> undergraduates;  // other information about graduate  };  class Teacher {  public:  std::string name;  Graduate graduate;  std::vector<Undergraduate> undergraduates;  void displayTeacherInfo() const {  std::cout << "Teacher: " << name << std::endl;  std::cout << "Graduate Student: " << graduate.name << std::endl;  std::cout << "Undergraduate Students: ";  for (const auto& undergrad : undergraduates) {  std::cout << undergrad.name << " ";  }  std::cout << std::endl;  }  };  class GeneralizedList {  private:  std::vector<Teacher> teachers;  public:  void insertTeacher(const Teacher& teacher) {  teachers.push\_back(teacher);  }  void deleteTeacher(const std::string& teacherName) {  for (auto it = teachers.begin(); it != teachers.end(); ++it) {  if (it->name == teacherName) {  teachers.erase(it);  break;  }  }  }  void queryTeacher(const std::string& teacherName) {  for (const auto& teacher : teachers) {  if (teacher.name == teacherName) {  teacher.displayTeacherInfo();  }  }  }  void outputList() {  for (const auto& teacher : teachers) {  teacher.displayTeacherInfo();  }  }  };  int main() {  // Test case  GeneralizedList list;  Teacher teacher1;  teacher1.name = "Dr. Smith";  Graduate graduate1;  graduate1.name = "Jack";  Undergraduate undergrad1;  undergrad1.name = "Alice";  graduate1.undergraduates.push\_back(undergrad1);  teacher1.graduate = graduate1;  Undergraduate undergrad2;  undergrad2.name = "Bob";  teacher1.undergraduates.push\_back(undergrad2);  Teacher teacher2;  teacher2.name = "Prof. Johnson";  Graduate graduate2;  graduate2.name = "Emily";  Undergraduate undergrad3;  undergrad3.name = "Carol";  graduate2.undergraduates.push\_back(undergrad3);  teacher2.graduate = graduate2;  Undergraduate undergrad4;  undergrad4.name = "David";  teacher2.undergraduates.push\_back(undergrad4);  list.insertTeacher(teacher1);  list.insertTeacher(teacher2);  list.outputList();  list.queryTeacher("Dr. Smith");  list.deleteTeacher("Prof. Johnson");  list.outputList();  return 0;  } |

|  |
| --- |
|  |
| 三、实验结果分析、思考题解答∶  稀疏矩阵运算应用  实验结果分析   1. **正确性验证**：    * 通过一系列测试用例验证稀疏矩阵类及其运算（加法和乘法）的正确性。    * 比较稀疏矩阵运算结果与预期结果，确保算法实现正确。 2. **性能分析**：    * 分析稀疏矩阵运算的时间复杂度。例如，稀疏矩阵加法和乘法的时间复杂度分别是多少？    * 比较稀疏矩阵运算与普通矩阵运算的性能差异，特别是在矩阵非常稀疏的情况下。 3. **内存使用分析**：    * 分析稀疏矩阵类在存储大量数据时的内存使用情况。    * 比较稀疏矩阵表示与普通矩阵表示的内存使用差异。 4. **边界条件测试**：    * 测试边界条件，如空矩阵、全零矩阵、矩阵维度不匹配等情况。    * 确保在这些情况下，算法能够正确处理并抛出适当的错误或警告。   本科生导师制问题  实验结果分析   1. **正确性验证**：    * 通过一系列测试用例验证广义表类的插入、删除、查询和输出功能的正确性。    * 比较实际输出与预期输出，确保算法实现正确。 2. **性能分析**：    * 分析插入、删除、查询和输出操作的时间复杂度。    * 比较不同操作在不同数据规模下的性能表现。 3. **内存使用分析**：    * 分析广义表类在存储大量数据时的内存使用情况。    * 比较广义表表示与普通列表表示的内存使用差异。 4. **边界条件测试**：    * 测试边界条件，如空表、重复插入、删除不存在的元素等情况。    * 确保在这些情况下，算法能够正确处理并抛出适当的错误或警告。 |
| 四、感想、体会、建议∶  稀疏矩阵运算应用  感想与体会   1. **数据结构选择的重要性**：稀疏矩阵的运算应用展示了选择合适的数据结构的重要性。通过使用三元组表示法，可以显著减少存储空间和提高运算效率。 2. **算法设计的复杂性**：稀疏矩阵的加法和乘法虽然基础，但实现起来需要考虑很多细节，如边界条件、矩阵维度匹配等，这体现了算法设计的复杂性。 3. **测试与验证的必要性**：通过设计多种测试用例，可以验证算法的正确性和鲁棒性，这对于确保程序的可靠性至关重要。   建议   1. **优化算法**：可以进一步优化稀疏矩阵的运算算法，例如使用更高效的数据结构（如十字链表）来提高运算速度。 2. **扩展功能**：可以增加更多的矩阵运算功能，如转置、求逆等，以满足更广泛的应用需求。 3. **用户界面**：可以开发一个简单的用户界面，使用户能够更方便地输入和查看稀疏矩阵及其运算结果。   本科生导师制问题  感想与体会   1. **数据组织的灵活性**：广义表的使用展示了数据组织的灵活性，能够有效地表示复杂的关系结构，如导师、研究生和本科生之间的关系。 2. **操作实现的复杂性**：插入、删除、查询和输出操作虽然基础，但实现起来需要考虑很多细节，如数据结构的嵌套、元素的唯一性等，这体现了操作实现的复杂性。 3. **测试与验证的必要性**：通过设计多种测试用例，可以验证算法的正确性和鲁棒性，这对于确保程序的可靠性至关重要。   建议   1. **优化数据结构**：可以进一步优化广义表的数据结构，例如使用更高效的存储方式来提高操作速度。 2. **扩展功能**：可以增加更多的操作功能，如修改元素、统计元素数量等，以满足更广泛的应用需求。 3. **用户界面**：可以开发一个简单的用户界面，使用户能够更方便地输入和查看广义表及其操作结果。 |
| 实验成绩∶  指导教师签名：  年 月 日 |