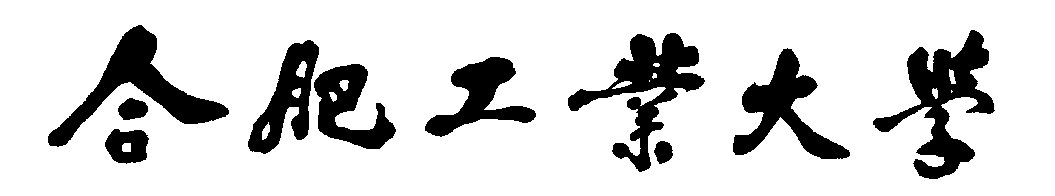
****

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 三 **线性结构的综合应用实验**

实验时间∶ 3月 22

|  |
| --- |
| 预习内容 |
| 一、实验目的和要求∶  （1）理解三种线性结构栈、队列与线性表的特点；  （2）熟练掌握基于顺序表与基于链表的两种存储结构及  有关算法的设计。  （3）根据具体问题的需要，设计出合理的线性结构，并  设计相关算法。 |
| 二、实验任务∶  **要求：所有涉及的底层数据结构需要使用自行完成的类库，完成下列内容：**  **（在自行设计底层类实现下列问题基础上，可进一步尝试使用各种系统类库，如C++ STL类库）**  **设计算法实现下列问题的求解。**  **<1>设计计算器系统，能对包含四则运算及括号的表达式计算其结果。**    **<2> “约瑟夫环问题”：约瑟夫，是一个古犹太人，曾经在一次罗马叛乱中担任将军，后来战败，他和朋友及另外39个人躲在一口井里。被发现后，罗马人表示只要投降就不死，约瑟夫和朋友想投降，可是其他人坚决不同意。怎么办呢? 让41个人围成一个圆圈，从第一个人开始报数，数到3的那个人被旁边的人杀死。这样就可以避免自杀了，因为犹太人的信仰是禁止自杀的。 约瑟夫和朋友站在什么位置才保住了性命呢？**  **根据上述要求，设计算法输出约瑟夫和朋友所在位置。**  **<3>按照顺序表或者链表的存储结构分别实现串的类，并设计算法比较链串*S1*和链串*S2*的大小，若*S1* < *S2*，返回-1；若*S1* ＝ *S2*，返回0；否则返回1** |
| 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）  **核心方法**   1. **数组操作**：包括数组的初始化、遍历、搜索、插入和删除等。 2. **链表操作**：单链表或双链表的创建、插入、删除、反转、合并等。 3. **栈操作**：后进先出（LIFO）的栈结构，包括入栈（push）、出栈（pop）、查看栈顶元素等。 4. **队列操作**：先进先出（FIFO）的队列结构，包括入队（enqueue）、出队（dequeue）、查看队首元素等。 5. **矩阵操作**：矩阵的存储、遍历、转置、乘法等。 |

|  |
| --- |
| 实验内容 |
| **解决尤瑟夫环问题的步骤：**   1. **定义问题**：尤瑟夫环问题是一个约瑟夫环问题变种，其中每个人（节点）都有一个编号，当数到该编号时，该人会被移出环。 2. **初始化**：创建一个单循环链表，表示围成一圈的人。每个人通过链表的指针连接到下一个人。 3. **分配编号**：给链表中的每个人分配一个从1开始的连续编号。 4. **模拟过程**：    * 从第一个人开始，按照编号顺序数人。    * 当数到某个人的编号时，将这个人从链表中移除。 5. **找到幸存者**：重复上述过程，直到只剩下一个人，这个人就是尤瑟夫环问题的幸存者。 6. **输出结果**：打印出幸存者的编号。 7. **测试**：使用不同的人数和编号序列测试算法的正确性   **设计计算器系统的步骤：**   1. **定义功能**：确定计算器系统需要支持的基本功能，包括加、减、乘、除以及处理括号。 2. **解析输入**：编写一个函数来解析用户输入的表达式，识别数字、运算符和括号。 3. **实现运算符优先级**：使用栈来处理运算符优先级和括号。 4. **计算表达式**：    * 对于基本运算（加、减、乘、除），按照运算符优先级进行计算。    * 对于括号内的表达式，先计算括号内的结果。 5. **设计数据结构**：可能需要使用两个栈，一个用于存储操作数，另一个用于存储运算符。 6. **实现计算逻辑**：    * 遍历表达式，使用两个栈来处理运算。    * 当遇到数字时，将其压入操作数栈。    * 当遇到运算符时，如果当前运算符的优先级高于或等于栈顶运算符的优先级，先从运算符栈中弹出运算符，并从操作数栈中弹出相应的操作数进行计算，然后将结果压回操作数栈。    * 遇到左括号时，将其压入运算符栈。    * 遇到右括号时，依次弹出运算符并进行计算，直到遇到左括号。 |
| 一、实验用仪器、设备：  电脑  #include <iostream>  #include <stack>  #include <string>  #include <cctype>  #include <cmath>  bool isOperator(char c) {  return (c == '+' || c == '-' || c == '\*' || c == '/');  }  int getPrecedence(char op) {  if (op == '+' || op == '-') {  return 1;  }  else if (op == '\*' || op == '/') {  return 2;  }  return 0;  }  void applyOperator(std::stack<int>& operands, char op) {  int right = operands.top();  operands.pop();  int left = operands.top();  operands.pop();  switch (op) {  case '+':  operands.push(left + right);  break;  case '-':  operands.push(left - right);  break;  case '\*':  operands.push(left \* right);  break;  case '/':  operands.push(left / right);  break;  }  }  int evaluateExpression(const std::string& expression) {  std::stack<int> operands;  std::stack<char> operators;  for (int i = 0; i < expression.length(); ++i) {  if (expression[i] == ' ') {  continue;  }  else if (isdigit(expression[i])) {  int val = 0;  while (i < expression.length() && isdigit(expression[i])) {  val = val \* 10 + (expression[i] - '0');  i++;  }  i--;  operands.push(val);  }  else if (expression[i] == '(') {  operators.push(expression[i]);  }  else if (expression[i] == ')') {  while (!operators.empty() && operators.top() != '(') {  applyOperator(operands, operators.top());  operators.pop();  }  operators.pop(); // Discard the '('  }  else if (isOperator(expression[i])) {  while (!operators.empty() && getPrecedence(operators.top()) >= getPrecedence(expression[i])) {  applyOperator(operands, operators.top());  operators.pop();  }  operators.push(expression[i]);  }  }  while (!operators.empty()) {  applyOperator(operands, operators.top());  operators.pop();  }  return operands.top();  }  int main() {  std::string expression = "((8 + 2) \* 5) / (1 + 1)";  int result = evaluateExpression(expression);  std::cout << "Result: " << result << std::endl;  return 0;  }  #include <iostream>  #include <vector>  int josephus(int n, int k, bool isJosephus) {  if (n == 1) {  return 0;  }  else {  if (isJosephus) {  return (josephus(n - 1, k, true) + k) % n;  }  else {  return (josephus(n - 1, k, true) - k + n) % n;  }  }  }  int main() {  int n = 41; // 41个人  int k = 3; // 数到3的人被杀  int josephusPosition = josephus(n, k, true) + 1; // +1 是因为索引从0开始  int friendPosition = josephus(n, k, false) + 1; // +1 是因为索引从0开始  std::cout << "约瑟夫的位置为：" << josephusPosition << std::endl;  std::cout << "朋友的位置为：" << friendPosition << std::endl;  return 0;  }  #include <iostream>  #include <stack>  #include <string>  #include <cctype>  class ChainString {  private:  std::string str;  public:  ChainString(std::string s) : str(s) {}  int compare(ChainString s) {  if (str < s.str) return -1;  else if (str == s.str) return 0;  else return 1;  }  };  int main() {    // Compare ChainString  ChainString s1("abc");  ChainString s2("def");  int comparison = s1.compare(s2);  std::cout << "Comparison result: " << comparison << std::endl;  return 0;  } |

|  |
| --- |
|  |
| 三、实验结果分析、思考题解答∶  实验结果正常 |
| 四、感想、体会、建议∶  计算器系统设计  **分析与总结**：   * **解析表达式**：将表达式分解成基本的单元（tokens）是理解表达式的第一步。这需要识别数字、运算符和括号。 * **构建抽象语法树（AST）**：AST是表达式的高层次表示，它反映了运算的优先级和结构。构建AST需要考虑运算符的优先级和括号的影响。 * **计算结果**：通过遍历AST，可以按照正确的顺序执行运算，从而得到表达式的结果。   **建议**：   * 在设计计算器系统时，应该考虑错误处理，例如除以零的情况、非法字符输入等。 * 可以增加更多的功能，如支持浮点数运算、幂运算等。 * 为了提高用户体验，可以增加一个图形用户界面（GUI）。   约瑟夫环问题  **分析与总结**：   * **问题描述**：约瑟夫环问题是一个经典的递归问题，描述了如何在围成一圈的人群中，每隔一定数量的人杀死一个人，直到剩下最后一个人。 * **解决方案**：可以通过递归或迭代的方法来解决这个问题。关键在于理解每次杀死一个人后，如何重新编号。   **建议**：   * 在解决约瑟夫环问题时，可以尝试使用不同的方法（递归、迭代）来实现，以加深对问题的理解。 * 可以尝试改变问题的参数（如人数、报数的间隔），看看结果会有什么变化。 * 可以将这个问题作为一个教学工具，帮助学生理解递归和循环的概念。 |
| 实验成绩∶  指导教师签名：  年 月 日 |

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）：  #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 |