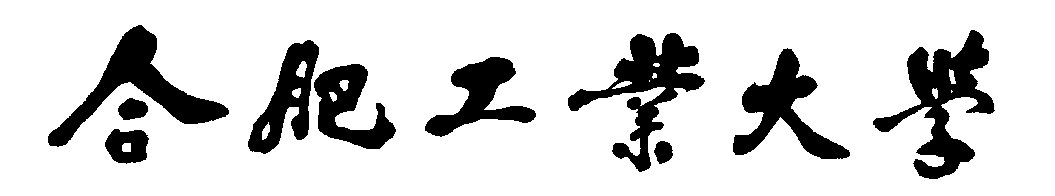
****

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 5 **树与二叉树的综合应用实验**

实验时间∶2024 年6 月1 日

|  |  |  |
| --- | --- | --- |
| 预习内容 | | |
| 一、实验目的和要求∶  **（1）掌握树与二叉树的动态链表存储结构及表示；**  **（2）掌握二叉树、线索二叉树、树与森林的遍历算法与相互转换；**  **（3）掌握基于树与二叉树的实际应用问题求解的算法设计** | | |
| 二、实验任务∶ |  | |
| 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. **二叉树节点定义**：创建一个二叉树节点类，包含数据、左子节点指针和右子节点指针。 2. **中序遍历**：实现中序遍历算法，同时记录节点的层次数。 3. **线索化算法**：实现前序、中序、后序线索化二叉树的算法。 4. **线索二叉树插入**：在线索二叉树的指定位置插入新节点。  框架  1. **定义二叉树节点类**：包含数据域和左右子节点指针域。 2. **定义二叉树类**：包含根节点指针和相关操作的方法。 3. **实现中序遍历**：记录并输出节点值和层次。 4. **实现线索化方法**：将二叉树转换为线索二叉树。 5. **实现插入方法**：在线索二叉树中插入节点。    核心方法  1. **树和森林的表示**：定义树和森林的数据结构，可能使用节点类和相关的连接方式。 2. **建树和森林**：实现多种建树或森林的方法，如指定输入、文件读入等。 3. **树遍历算法**：实现前序、中序、后序遍历算法。 4. **树和森林的转换**：实现树到二叉树和二叉树到树的转换算法。  框架  1. **定义树节点类**：包含数据域和子节点指针域。 2. **定义树和森林类**：包含树的根节点和相关操作的方法。 3. **实现建树方法**：提供多种方式构建树或森林。 4. **实现遍历方法**：提供树的多种遍历方式。 5. **实现转换方法**：实现树与二叉树之间的转换。 | | |
| 问题2：树和森林的小系统核心方法  1. **树和森林的表示**：定义树和森林的数据结构，可能使用节点类和相关的连接方式。 2. **建树和森林**：实现多种建树或森林的方法，如指定输入、文件读入等。 3. **树遍历算法**：实现前序、中序、后序遍历算法。 4. **树和森林的转换**：实现树到二叉树和二叉树到树的转换算法。  框架  1. **定义树节点类**：包含数据域和子节点指针域。 2. **定义树和森林类**：包含树的根节点和相关操作的方法。 3. **实现建树方法**：提供多种方式构建树或森林。 4. **实现遍历方法**：提供树的多种遍历方式。 5. **实现转换方法**：实现树与二叉树之间的转换。  流程图 | |  |
| **问题三**  **核心方法：**   1. **字符权重计算**：读取文件中的每个字符，并统计每个字符出现的次数，以此作为权重。 2. **哈夫曼树构建**：利用字符权重构建哈夫曼树。这通常涉及以下步骤：    * 创建一个优先队列（最小堆），将每个字符作为叶子节点加入队列。    * 重复从队列中取出两个权重最小的节点，创建一个新的内部节点，其权重为这两个节点权重之和，然后将这个新节点加入队列。    * 重复上述步骤，直到队列中只剩下一个节点，这个节点即为哈夫曼树的根节点。 3. **生成哈夫曼编码**：从哈夫曼树的根节点开始，为每个字符生成唯一的哈夫曼编码。左分支用"0"表示，右分支用"1"表示。  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）  **实验一：二叉树系统设计**   1. **定义二叉树节点**：创建一个二叉树节点类，包含数据、左子节点指针和右子节点指针。 2. **实现二叉树类**：创建二叉树类，包含添加节点、删除节点等基本操作。 3. **中序遍历并输出层次**：    * 实现中序遍历算法，同时记录并输出每个节点的值和层次数。 4. **实现线索化算法**：    * 按照先序、中序、后序遍历的顺序，实现三种次序的线索化算法。 5. **在线索二叉树中插入节点**：    * 根据用户输入，在线索二叉树的指定位置插入新节点。 6. **测试案例**：    * 构建示例二叉树，验证中序遍历输出、线索化和插入节点的功能。   **实验二：树与森林的综合应用**   1. **构建树或森林**：    * 实现多种建树方法，如指定输入或从文件读入。 2. **实现遍历算法**：    * 实现前序、中序、后序遍历算法，并应用于森林。 3. **森林的后序遍历**：    * 根据示例森林，实现并测试后序遍历算法。 4. **树与森林的转换**：    * 实现将森林转换为二叉树的算法，并验证转换结果。 5. **测试案例**：    * 准备测试数据，验证建树、遍历和转换功能的正确性。   **实验三：压缩与解压缩系统设计**   1. **读取文件和权重计算**：    * 读取文件内容，统计并计算每个字符的权重。 2. **设计哈夫曼树**：    * 根据字符权重，构建哈夫曼树，并输出每个字符的哈夫曼编码。 3. **文件压缩**：    * 使用哈夫曼编码对文件进行压缩，并存储压缩结果。    * **4.结果分析**    * 结果测试准备好文件，看程序是否生成压缩好的文件   ：  #include <iostream>  #include <vector>  // 定义二叉树节点结构  struct TreeNode {  char value;  int level; // 节点所在的层次  TreeNode\* left;  TreeNode\* right;  TreeNode(char val) : value(val), level(0), left(nullptr), right(nullptr) {}  };  // 构建二叉树并记录每个节点的层次数  void buildTreeWithLevel(const std::vector<std::pair<char, std::pair<int, int>>>& nodes, std::vector<TreeNode\*>& treeNodes, TreeNode\* root, int level) {  if (root == nullptr) return;  root->level = level;  if (nodes[root->value - 'A'].second.first != -1) {  root->left = treeNodes[nodes[root->value - 'A'].second.first];  buildTreeWithLevel(nodes, treeNodes, root->left, level + 1);  }  if (nodes[root->value - 'A'].second.second != -1) {  root->right = treeNodes[nodes[root->value - 'A'].second.second];  buildTreeWithLevel(nodes, treeNodes, root->right, level + 1);  }  }  // 中序遍历输出节点值及对应的层次数  void inorderTraversalWithLevel(TreeNode\* root, std::vector<std::pair<char, int>>& result) {  if (root == nullptr) return;  inorderTraversalWithLevel(root->left, result);  result.push\_back(std::make\_pair(root->value, root->level));  inorderTraversalWithLevel(root->right, result);  }  // 前序遍历  void preorderTraversal(TreeNode\* root) {  if (root == nullptr) return;  std::cout << root->value << " ";  preorderTraversal(root->left);  preorderTraversal(root->right);  }  // 后序遍历  void postorderTraversal(TreeNode\* root) {  if (root == nullptr) return;  postorderTraversal(root->left);  postorderTraversal(root->right);  std::cout << root->value << " ";  }  // 在指定位置插入节点  void insertNode(TreeNode\* root, char parentValue, char newValue, bool isLeft) {  if (root == nullptr) return;  if (root->value == parentValue) {  if (isLeft) {  TreeNode\* newNode = new TreeNode(newValue);  newNode->left = root->left;  root->left = newNode;  }  else {  TreeNode\* newNode = new TreeNode(newValue);  newNode->right = root->right;  root->right = newNode;  }  return;  }  insertNode(root->left, parentValue, newValue, isLeft);  insertNode(root->right, parentValue, newValue, isLeft);  }  int main() {  // 构建二叉树  std::vector<std::pair<char, std::pair<int, int>>> nodes = {  {'A', {1, 2}}, {'B', {3, 4}}, {'C', {5, 6}}, {'D', {-1, -1}}, {'E', {-1, -1}}, {'F', {-1, -1}}, {'G', {-1, -1}}  };  std::vector<TreeNode\*> treeNodes(nodes.size(), nullptr);  for (size\_t i = 0; i < nodes.size(); ++i) {  treeNodes[i] = new TreeNode(nodes[i].first);  }  for (size\_t i = 0; i < nodes.size(); ++i) {  if (nodes[i].second.first != -1) {  treeNodes[i]->left = treeNodes[nodes[i].second.first];  }  if (nodes[i].second.second != -1) {  treeNodes[i]->right = treeNodes[nodes[i].second.second];  }  }  buildTreeWithLevel(nodes, treeNodes, treeNodes[0], 1);  // 中序遍历输出节点值及对应的层次数  std::vector<std::pair<char, int>> result;  inorderTraversalWithLevel(treeNodes[0], result);  std::cout << "中序遍历输出节点值及对应的层次数: ";  for (const auto& p : result) {  std::cout << p.first << ":" << p.second << ", ";  }  std::cout << std::endl;  // 遍历二叉树  std::cout << "前序遍历: ";  preorderTraversal(treeNodes[0]);  std::cout << std::endl;  // 插入新节点  insertNode(treeNodes[0], 'D', 'H', true);  // 遍历二叉树以验证插入结果  std::cout << "插入新节点后的前序遍历: ";  preorderTraversal(treeNodes[0]);  std::cout << std::endl;  return 0;  }  #include <iostream>  #include <vector>  #include <fstream>  using namespace std;  struct TreeNode {  char val;  vector<TreeNode\*> children;  TreeNode(char x) : val(x) {}  };  struct Forest {  vector<TreeNode\*> trees;  };  struct BinaryTreeNode {  char val;  BinaryTreeNode\* left;  BinaryTreeNode\* right;  BinaryTreeNode(char x) : val(x), left(nullptr), right(nullptr) {}  };  void displayMenu() {  cout << "1. 采用指定输入建树或森林" << endl;  cout << "2. 读入文件建树或森林" << endl;  cout << "3. 实现各遍历算法" << endl;  cout << "4. 与二叉树的相互转换" << endl;  cout << "5. 退出" << endl;  }  void buildSubTree(TreeNode\* node) {  int numChildren;  cout << "输入节点 " << node->val << " 的子节点数量: ";  cin >> numChildren;  for (int i = 0; i < numChildren; ++i) {  char childVal;  cout << "输入子节点 " << i + 1 << " 的值: ";  cin >> childVal;  TreeNode\* child = new TreeNode(childVal);  node->children.push\_back(child);  buildSubTree(child);  }  }  void buildSubTreeFromFile(TreeNode\* node, ifstream& file) {  int numChildren;  file >> numChildren;  for (int i = 0; i < numChildren; ++i) {  char childVal;  file >> childVal;  TreeNode\* child = new TreeNode(childVal);  node->children.push\_back(child);  buildSubTreeFromFile(child, file);  }  }  TreeNode\* buildTreeFromFile(ifstream& file) {  char rootVal;  file >> rootVal;  TreeNode\* root = new TreeNode(rootVal);  // 递归构建子树  buildSubTreeFromFile(root, file);  return root;  }  TreeNode\* buildTreeFromInput() {  char rootVal;  cout << "输入根节点的值: ";  cin >> rootVal;  TreeNode\* root = new TreeNode(rootVal);  // 递归构建子树  buildSubTree(root);  return root;  }  void postOrderTraversal(TreeNode\* node) {  if (node) {  for (TreeNode\* child : node->children) {  postOrderTraversal(child);  }  cout << node->val;  }  }  BinaryTreeNode\* convertToBinaryTree(TreeNode\* node) {  if (!node) return nullptr;  BinaryTreeNode\* bNode = new BinaryTreeNode(node->val);  if (!node->children.empty()) {  bNode->left = convertToBinaryTree(node->children[0]);  }  BinaryTreeNode\* current = bNode->left;  for (int i = 1; i < node->children.size(); ++i) {  current->right = convertToBinaryTree(node->children[i]);  current = current->right;  }  return bNode;  }  void preOrderTraversal(BinaryTreeNode\* node) {  if (node) {  cout << node->val;  preOrderTraversal(node->left);  preOrderTraversal(node->right);  }  }  void inOrderTraversal(BinaryTreeNode\* node) {  if (node) {  inOrderTraversal(node->left);  cout << node->val;  inOrderTraversal(node->right);  }  }  void postOrderTraversal(BinaryTreeNode\* node) {  if (node) {  postOrderTraversal(node->left);  postOrderTraversal(node->right);  cout << node->val;  }  }  int main() {  int choice;  Forest forest;  while (true) {  displayMenu();  cout << "选择一个选项: ";  cin >> choice;  switch (choice) {  case 1: {  TreeNode\* tree = buildTreeFromInput();  forest.trees.push\_back(tree);  break;  }  case 2: {  ifstream file("tree\_data.txt");  if (file.is\_open()) {  TreeNode\* tree = buildTreeFromFile(file);  forest.trees.push\_back(tree);  file.close();  }  else {  cout << "无法打开文件" << endl;  }  break;  }  case 3: {  cout << "后序遍历森林: ";  for (TreeNode\* tree : forest.trees) {  postOrderTraversal(tree);  }  cout << endl;  break;  }  case 4: {  cout << "转换为二叉树并遍历: " << endl;  for (TreeNode\* tree : forest.trees) {  BinaryTreeNode\* bTree = convertToBinaryTree(tree);  cout << "前序遍历: ";  preOrderTraversal(bTree);  cout << endl;  cout << "中序遍历: ";  inOrderTraversal(bTree);  cout << endl;  cout << "后序遍历: ";  postOrderTraversal(bTree);  cout << endl;  }  cout << endl;  break;  }  case 5: {  return 0;  }  default: {  cout << "无效选项" << endl;  break;  }  }  }  return 0;  }  #include <iostream>  #include <fstream>  #include <queue>  #include <unordered\_map>  #include <vector>  #include <string>  using namespace std;  // 哈夫曼树节点  struct HuffmanNode {  char data;  int frequency;  HuffmanNode\* left, \* right;  HuffmanNode(char data, int frequency) : data(data), frequency(frequency), left(nullptr), right(nullptr) {}  };  // 用于优先队列的比较函数  struct Compare {  bool operator()(HuffmanNode\* a, HuffmanNode\* b) {  return a->frequency > b->frequency;  }  };  // 生成哈夫曼树  HuffmanNode\* buildHuffmanTree(const unordered\_map<char, int>& frequencyMap) {  priority\_queue<HuffmanNode\*, vector<HuffmanNode\*>, Compare> minHeap;  for (const auto& pair : frequencyMap) {  minHeap.push(new HuffmanNode(pair.first, pair.second));  }  while (minHeap.size() != 1) {  HuffmanNode\* left = minHeap.top();  minHeap.pop();  HuffmanNode\* right = minHeap.top();  minHeap.pop();  HuffmanNode\* newNode = new HuffmanNode('$', left->frequency + right->frequency);  newNode->left = left;  newNode->right = right;  minHeap.push(newNode);  }  return minHeap.top();  }  // 生成哈夫曼编码  void generateHuffmanCodes(HuffmanNode\* root, string code, unordered\_map<char, string>& huffmanCodes) {  if (root == nullptr) return;  if (root->data != '$') {  huffmanCodes[root->data] = code;  }  generateHuffmanCodes(root->left, code + "0", huffmanCodes);  generateHuffmanCodes(root->right, code + "1", huffmanCodes);  }  // 压缩文件  void compressFile(const string& inputFile, const string& outputFile, const unordered\_map<char, string>& huffmanCodes) {  ifstream inFile(inputFile, ios::in | ios::binary);  ofstream outFile(outputFile, ios::out | ios::binary);  char ch;  string encodedString = "";  while (inFile.get(ch)) {  encodedString += huffmanCodes.at(ch);  }  // 将编码后的字符串写入输出文件  string compressedString = "";  int i = 0;  while (i < encodedString.size()) {  unsigned char byte = 0;  for (int j = 0; j < 8 && i < encodedString.size(); ++j, ++i) {  byte <<= 1;  if (encodedString[i] == '1') {  byte |= 1;  }  }  compressedString += byte;  }  outFile.write(compressedString.c\_str(), compressedString.size());  inFile.close();  outFile.close();  }  int main() {  string inputFile = "input.txt";  string outputFile = "compressed.bin";  // 读取文件并计算字符的权重  ifstream inFile(inputFile, ios::in | ios::binary);  unordered\_map<char, int> frequencyMap;  char ch;  while (inFile.get(ch)) {  frequencyMap[ch]++;  }  inFile.close();  // 生成哈夫曼树  HuffmanNode\* root = buildHuffmanTree(frequencyMap);  // 生成哈夫曼编码  unordered\_map<char, string> huffmanCodes;  generateHuffmanCodes(root, "", huffmanCodes);  // 输出每个字符的哈夫曼编码  for (const auto& pair : huffmanCodes) {  cout << pair.first << ": " << pair.second << endl;  }  // 压缩文件  compressFile(inputFile, outputFile, huffmanCodes);  cout << "文件压缩完成，输出文件为 " << outputFile << endl;  return 0;  }      压缩后 | | |

|  |
| --- |
|  |
| 三、实验结果分析、思考题解答∶   1. **正确性验证**：    * 检查所有实现的算法（如树的遍历、二叉树的遍历、树与二叉树的转换等）是否返回预期的结果。 2. **性能评估**：    * 分析算法的时间复杂度和空间复杂度。使用不同大小的数据集测试算法，观察运行时间和内存使用情况。 3. **边界条件测试**：    * 确保算法能够处理各种边界情况，例如空树、只有一个节点的树、所有节点都只有左（或右）子节点的树等。 4. **异常处理**：    * 分析算法在遇到非法输入或异常情况时的表现，确保有适当的错误处理和恢复机制。 5. **用户界面和体验**：    * 如果实验包括用户界面，评估其易用性、交互性和用户满意度。 6. **代码质量**：    * 检查代码的可读性、可维护性和扩展性。确保代码风格一致，有适当的注释和文档。 7. **实验对比**：    * 如果有多种算法实现同一功能，比较它们的效率和效果，分析各自优缺 |
| 四、感想、体会、建议∶  感想与体会   1. **数据结构的重要性**：树和二叉树是计算机科学中非常重要的数据结构，它们在许多算法和应用中都有广泛的应用。通过实验，可以深刻理解这些数据结构的特性和用途。 2. **算法设计的复杂性**：树和二叉树的操作，如遍历、查找、插入和删除，虽然基础，但实现起来需要考虑很多细节，如平衡性、节点关系等，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. **深入理解概念**：在实验前，确保对树和二叉树的基本概念有深入的理解，如树的高度、节点的度、二叉树的遍历方式等。 2. **优化算法**：可以尝试优化树和二叉树的操作算法，例如使用平衡二叉树（如AVL树、红黑树）来提高查找和插入的效率。 3. **扩展实验内容**：可以增加更多的实验内容，如实现堆、B树、B+树等更复杂的数据结构，以加深对树结构的理解。 4. **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树及其操作结果。 5. **性能分析**：在实验过程中，可以对不同操作进行性能分析，比较不同数据结构和算法的效率，以加深对性能优化的理解 |
| 实验成绩∶  指导教师签名：  年 月 日 |