**数据结构实验报告**

|  |  |  |  |
| --- | --- | --- | --- |
| 学号-姓名 |  | 实验时间 | 2023年 10月 26日 |
| 诚信声明 | 手写，签名 | | |
| 实验题目 | 二叉树的遍历运算 | | |
| 实验过程中遇到的主要问题 | 先序遍历、中序遍历和后序遍历的实现：我同时实现了递归与非递归形式。在非递归形式中，需要使用栈等数据结构来辅助遍历过程。 | | |
| 实验小结 | 本次实验主要围绕二叉树的遍历运算展开，通过实现先序遍历、中序遍历和后序遍历，对二叉树的遍历方式有了更深入的理解。在实现过程中，需要注意节点的创建和关系的建立，以及如何使用栈等数据结构来辅助遍历。同时，实验还加深了我对二叉树的理解和对数据结构的运用能力。 | | |
| 数据结构  （自定义数据类型） | // 树的数据结构  typedef char TElemType;  typedef struct BiTNode \*BiTree;  struct BiTNode {  TElemType Data;  BiTree Left, Right; // 左右子树  int flag; // 用于非递归遍历  }；  // 栈的数据结构  typedef BiTNode \*SElemType;  typedef struct SNode \*PSNode, \*Stack;  struct SNode {  SElemType Data;  PSNode Next;  }; | | |
| 主要算法  （或算法说明） | 创建二叉树：  根据输入序列，逐个创建节点，并根据节点的值和位置关系构建节点之间的连接。用”.”表示节点为空  先序遍历运算：   1. 如果当前节点为空，则返回。 2. 访问当前节点。 3. 递归地对当前节点的左子树进行先序遍历。 4. 递归地对当前节点的右子树进行先序遍历。   中序遍历运算：   1. 如果当前节点为空，则返回。 2. 递归地对当前节点的左子树进行中序遍历。 3. 访问当前节点。 4. 递归地对当前节点的右子树进行中序遍历。   后序遍历运算：   1. 如果当前节点为空，则返回。 2. 递归地对当前节点的左子树进行后序遍历。 3. 递归地对当前节点的右子树进行后序遍历。 4. 访问当前节点。   先序遍历（非递归）：   1. 创建一个空栈，将根节点入栈。 2. 循环执行以下步骤直到栈为空： 3. 弹出栈顶节点，如果当前节点为空，则跳过此次循环，否则访问该节点。 4. 如果该节点的右子节点不为空，将右子节点入栈。 5. 如果该节点的左子节点不为空，将左子节点入栈。   中序遍历（非递归）：   1. 创建一个空栈，将根节点入栈。 2. 循环执行以下步骤直到栈为空： 3. 循环执行以下步骤直到栈顶节点为空： 4. 获取栈顶结点，作为当前节点 5. 将左子节点入栈。 6. 如栈不为空，弹出栈顶节点，访问该节点，并将右子节点入栈。   后序遍历（非递归）：   1. 如果传入的二叉树为空，则直接返回。 2. 创建一个空栈，用于存储待访问的二叉树节点。 3. 将二叉树根节点的标记设置为0，表示未访问过。 4. 将二叉树根节点压入栈s中。 5. 进入循环，直到栈s为空：   a. 弹出栈顶元素并将其作为当前节点。如果当前节点为空，则跳过当前循环，继续下一次循环。  b. 根据当前节点的标记执行相应的操作：   * + 当标记为0时：     - 将标记设置为1，表示左子树已经访问过。     - 将当前节点重新压入栈中。     - 如果当前节点的左子树存在：     - 将当前节点的左子树的标记设置为0，表示未访问过。     - 将当前节点的左子树压入栈中。   + 当标记为1时：     - 将标记设置为2，表示右子树已经访问过。     - 将当前节点重新压入栈中。     - 如果当前节点的右子树存在：     - 将当前节点的右子树的标记设置为0，表示未访问过。     - 将当前节点的右子树压入栈中。   + 当标记为2时：     - 访问当前节点 | | |

**数据结构实验报告**

|  |  |  |  |
| --- | --- | --- | --- |
| 学号-姓名 |  | 实验时间 | 2023年 10月 26日 |
| 诚信声明 | 手写，签名 | | |
| 实验题目 | 哈夫曼编/译码器 | | |
| 实验过程中遇到的主要问题 | 哈夫曼树的构建：需要根据输入的字符集和权值构建哈夫曼树。如何正确选择权值最小的两个节点构建新的节点较为复杂。  编码过程：对输入的文本进行编码，将字符映射为对应的哈夫曼编码。需要正确查询到字符所对应的编码。  译码过程：对编码后的数据进行译码，将01序列翻译为原始文本。同样需要用正确的顺序访问哈夫曼树。 | | |
| 实验小结 | 本实验实现了一个基于哈夫曼编码的编/译码器。通过构建哈夫曼树，将输入的文本进行编码，并将编码结果保存到文件中。然后通过读取编码结果文件，进行译码操作，将编码的01序列翻译为原始文本。实验中主要涉及到哈夫曼树的构建、编码过程、译码过程以及文件操作。  在实现过程中，需要设计合适的数据结构来表示哈夫曼树，并实现对应的算法。可以使用数组、链表等数据结构来表示树的节点和编码结果的存储。同时，还需要考虑字符集和权值的输入、保存哈夫曼树、编码结果和译码结果的文件操作等。  通过本实验，我加深了对哈夫曼编码的理解，掌握了哈夫曼编/译码的基本原理和实现方法。同时也熟悉了文件操作的相关知识和技巧。实验中遇到的问题主要集中在数据结构的设计和算法的实现上，通过仔细分析问题、查找相关资料和调试代码，最终成功实现了哈夫曼编/译码器。 | | |
| 数据结构  （自定义数据类型） | // 树的数据结构  struct HTNode {  char ch;  double weight;  int parent, lchild, rchild;  };  typedef HTNode \*HuffmanTree;  struct CharCode { // 字符和对应的01序列  char ch;  char \*code;  };  typedef CharCode \*HuffmanCode; | | |
| 主要算法  （或算法说明） | 初始化算法：   * 从终端（键盘）读入字符集的大小n，n个字符及n个权值，存储为节点列表。 * 每次选择权值最小的，且没有父亲节点的两个节点，构建新的节点（权值为两个节点的和）并加入列表。 * 利用构建好的树，通过遍历的方式将其处理为编码列表，具体方式为将左节点编码为0，右节点编码为1，碰到叶子节点就存储编码。以便于后续查找。 * 将建立好的哈夫曼树存储在文件hfmTree中。   编码算法：   * 对文本文件ToBeTran中的正文进行编码,利用已经建好的哈夫曼编码列表，对每个字符查询编码并组合起来。 * 将结果存入文件CodeFile中。   译码算法：   * 利用已经建好的哈夫曼树，对文本文件CodeFile中的代码进行翻译, 具体方式为将0编码为左节点，1编码为右节点，碰到叶子节点就存储字符并重新开始新的一轮。 * 将译码结果存入文件TextFile中。   比较文件算法：   * 读入并比较文件ToBeTran和TextFile的内容是否相同，若不同，则编译码过程中存在错误，需要进行修复。 | | |