四川大学计算机学院 （软件学院、 智能科学与技术学院）

实验报告

学号：2x 姓名： x 专业： 人工智能 班级： 233040701 第 13周

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 课程名称 | 数据结构与算法分析课程设计 | | 实验课时 | 4课时 |
| 实验题目 | 压缩软件 | | 实验时间 | 2024/11/14 |
| 实验目的和要求 | **基本要求**   1. 利用哈夫曼编码实现压缩与解压功能 2. 编码和译码的效率尽可能高   **已实现的扩展要求**   1. 替换CharString，提高效率 2. 压缩文件时先一次性将文件读入内存 | | | |
| 实验环境 | Clion2024、MinGW11.0、C++17 | | | |
| 算法描述 | **总体概览**  由于我不希望在在哈夫曼编码中引入字符串来存储0或1，于是我就这一部分重写了，没有使用事先提供的源码。  由于使用别的语言比较多，用byte类型用习惯了，于是使用宏定义#define byte char，也就是说，下文中的byte就是char  **对于哈夫曼编码**   1. 定义哈夫曼树节点结构体 HuffmanNode：   byte data：存储节点的数据，这里用 byte 类型表示。  int freq：存储节点的计数，用于构建哈夫曼树时的比较。  HuffmanNode \*left, \*right：指向左右子节点的指针。   1. 定义比较器结构体 compare：   bool operator()(HuffmanNode\* l, HuffmanNode\* r)：重载 () 运算符，用于比较两个哈夫曼节点的频率。如果左节点的频率大于右节点的频率，则返回 true，否则返回 false。这个比较器用于构建最小堆。   1. 定义哈夫曼树类 HuffmanTree：   私有构造函数：HuffmanTree() 被标记为 delete，这阻止了类的实例化。因为这个类只是一个工具类，我只希望使用该类的静态方法，对于这个类的实例化是毫无意义的。  静态方法 buildHuffmanTree()，接受byte与计数作为参数，返回根节点指针：  利用stl容器中的priority\_queue创建一个最小堆minHeap，用于存储哈夫曼节点。  for (int i = 0; i < data.size(); ++i)：遍历数据和频率数组，将每个数据元素及其频率作为一个哈夫曼节点插入到最小堆中。  while (minHeap.size()!= 1)：当最小堆中不止一个节点时，执行循环。  HuffmanNode \*left = minHeap.top(); minHeap.pop();：从最小堆中取出频率最小的节点作为左子节点，并从堆中删除。  HuffmanNode \*right = minHeap.top(); minHeap.pop();：从最小堆中取出频率次小的节点作为右子节点，并从堆中删除。  auto \*top = new HuffmanNode((byte)0, left->freq + right->freq);：创建一个新的哈夫曼节点，其频率为左右子节点频率之和。  top->left = left; top->right = right;：将新节点的左右子节点设置为刚才取出的左右子节点。  minHeap.push(top);：将新节点插入到最小堆中。  循环执行以上内容。  return minHeap.top();：循环结束后，最小堆中只剩下一个节点，即构建好的哈夫曼树的根节点，返回这个节点。  通过这个算法，可以根据给定的数据和频率构建出一棵哈夫曼树，用于后续的编码和解码操作。  **比特流辅助类**  定义了一个名为 BitInputBuffer 的类，它用于处理生成的比特流。这个类的设计目的是提供一个高效的比特流操作缓冲区，它可以在比特级别上进行操作。在哈夫曼编码中，需要根据编码规则在比特流中插入特定的比特值，或读取比特进行解码，利用此比特辅助类可以大幅增强代码的可读性，利于维护。  以下是这个算法原理：  构造函数 BitInputBuffer(int len)：  初始化一个指定长度的字节数组 p，用于存储输入的比特流。  设置当前指针 current 指向数组的起始位置。  初始化偏移量 offset 为 0，表示当前比特在字节中的位置。   * 1. **成员函数 push0()：**   这个函数用于在当前指针与offset位置写入一个 0 比特。  首先，它将当前字节 \*current 与一个掩码 ~(1 << offset) 进行按位与操作，将当前偏移量位置的比特清零。  然后，根据偏移量的值更新当前指针的位置：  如果偏移量等于 7（即当前字节的最后一个比特），则将偏移量重置为 0，并将当前指针移动到下一个字节。  否则，将偏移量加 1，表示下一个要写入的比特位置。   * 1. **成员函数 push1()：**   这与push0类似，不再过多赘述。  与之对应，BitOutputBuffer 类是一个用于处理比特流读取的缓冲区。它的主要功能是从一个字节数组中读取比特，并提供一个接口来获取下一个可用的比特   * 1. 构造函数 BitOutputBuffer(vector<byte> source)：   初始化一个字节数组 source，用于存储比特流数据。  使用 std::move 来转移 source 向量的所有权，以避免不必要的拷贝。   * 1. 成员函数 get()：   返回下一个可用的比特。  首先，从 source 数组中获取当前索引 index 处的字节 b。  然后，使用位操作 & 和 >> 来提取 b 中 offset 位置的比特，并将其存储在 v 中。  如果 offset 等于 7，表示已经读取了当前字节的最后一个比特，此时将 offset 重置为 0，并将 index 加 1，指向下一个字节。  否则，将 offset 加 1，表示下一个要读取的比特位置。  最后，返回提取的比特 v。  **哈夫曼编码的封装类** 这个代码定义了一个名为 Huffman 的类，用于实现 Huffman 编码和解码。是对上文的哈夫曼树的一个封装，利用比特流辅助类，实现了编码解码接口：   1. **宏定义**：    * byte 被定义为 char 类型。    * HuffmanCode 被定义为 vector<int> 类型。 2. **类定义**：    * Huffman 类包含两个受保护的成员变量：leafNodes（一个 map，用于存储每个字节及其对应的 Huffman 编码）和 treeTop（指向 Huffman 树顶部的指针）。 3. **受保护的成员函数**：    * encodingHuffman：这是一个递归函数，用于从 Huffman 树中生成 Huffman 编码。它接受一个 Huffman 树节点和一个 Huffman 编码作为参数。如果节点是抽象节点（即内部节点），则递归地为左右子节点生成编码；如果节点是叶子节点，则将其数据及其对应的编码存储在 leafNodes 中。 4. **构造函数**：    * Huffman()：默认构造函数。 5. **公共成员函数**：    * fromWeights：这是一个静态函数，用于根据给定的数据和权重构建 Huffman 树，并生成 Huffman 编码。它接受两个参数：data（数据向量）和 weights（权重向量）。它首先使用 HuffmanTree::buildHuffmanTree 函数构建 Huffman 树，然后使用 encodingHuffman 函数生成 Huffman 编码，并返回一个 Huffman 对象。    * encoding：用于对给定的数据进行 Huffman 编码。它接受一个 source（数据向量）作为参数，并返回编码后的数据。它首先计算编码的总长度，然后使用 BitInputBuffer 将编码后的数据存储在缓冲区中，最后将缓冲区中的数据转换为字节向量并返回。    * decoding：用于对给定的数据进行 Huffman 解码。它接受两个参数：source（编码后的数据向量）和 length（解码后的数据长度）。它使用 BitOutputBuffer 从编码后的数据中读取位，并根据 Huffman 树进行解码，最后将解码后的数据存储在 decoded 向量中并返回。    * ~Huffman()：析构函数，用于销毁 Huffman 树。   这个类实现了 Huffman 编码和解码的基本功能。提供了编码、解码api.  **压缩文件封装**  **decompress 函数**   1. **输入**: 一个字节数组 raw，其中包含了压缩的数据。 2. **输出**: 解压缩后的原始数据。   步骤:   * 从 raw 的前四个字节中读取元数据的长度（metaLength）。 * 从 raw 的接下来的四个字节中读取内容的长度（length）。 * 计算 Huffman 树的数据和权重的长度（huffmanLen）。 * 从 raw 中提取 Huffman 树的数据（data）和权重（weights）。 * 从 raw 中提取压缩后的内容（content）。 * 使用 Huffman::fromWeights 方法创建一个 Huffman 编码器，并使用 decoding 方法解压缩 content。 * 返回解压缩后的原始数据。   **compress 函数**   1. **输入**: 一个字节数组 raw，其中包含了原始数据。 2. **输出**: 压缩后的数据。   步骤:   * 统计 raw 中每个字节的出现频率，并存储在 map 中。 * 从 map 中提取字节值（data）和对应的频率（weights）。 * 使用 Huffman::fromWeights 方法创建一个 Huffman 编码器。 * 使用 Huffman 编码器的 encoding 方法压缩 raw。 * 构建压缩后的结果，包括元数据长度、内容长度、Huffman 树的数据和权重，以及压缩后的内容。   这两个函数再次封装了上述编码算法，为文件加入了元信息(metadata)。 | | | |
| 源程序清单 | main.cpp 定义程序入口  Compressor.h 定义压缩/解压类  Huffman.h 哈夫曼编码/解码的封装类  HuffmanTree.h 定义哈夫曼树与节点  BitBuffer.h 定义比特流  Utils.h 定义工具类：文本文件读写等 | | | |
| 运行结果 | 测试用例使用了一个LaTeX源代码，    下图是文件大小：源文件（左），压缩后（中），解压后（右），计算可得压缩率为78.1%    下图是源文件（左）与解压缩文件（右）的内容比对。 | | | |
| 实验运行情况分析(包括算法、运行结果、运行环境等问题的讨论) | **实验结果：**  实验最终运行结果与预期相符。  **问题的讨论：**  在编码解码过程中，示例程序使用了字符串为单位进行存储二进制的操作，我认为这非常浪费内存空间，并且浪费计算资源（判断相等条件等等）。因此，我想到了直接使用比特流，发现使用比特程序逻辑也更清晰了。  除此之外，在实现meta信息的编码时对指针的操作也有了更深刻的认识。 | | | |
| 指导老师  评议 |  | | | |
| 成绩评定： | 指导教师签名： | | |