**计算机科学与工程学院实验报告（1-4）**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **实验课程名称** | | **树和图应用类实验题目** | | | **实验成绩** |  |
| **专业** | **计算机科学与技术** | | **班级** | **计算机2201** | **指导教师签字** |  |
| **姓名** | **周小丁** | | **姓名** | **程文静** | **姓名** | **姚家路** |
| **学号** | **20225817** | | **学号** | **20222377** | **学号** | **20225973** |
| **附录实验程序及结果：**   1. **源程序**   附文件下载地址：https://github.com/lzysbl/huffmanTree  #include<iostream>  #include <fstream>  #include <string>  #include<opencv2/opencv.hpp>  #include <iosfwd>  #include <bitset>  #include <sstream>  typedef struct HafuTree{  int weight;  float data;  HafuTree\* LeftChild;  HafuTree\* RightChild;  HafuTree\* Parent;  }\*HuffTree,HTNode;  //创建哈夫曼树  void createHafuTree(HuffTree &root){  int m;  std::cout<<"请输入叶子节点的个数：";  std::cin>>m;  root = new HTNode[2\*m-1];  //初始化叶子节点  for(int i=0;i<m;i++){  std::cout<<"请输入第"<<i+1<<"个叶子节点的权值：";  std::cin>>root[i].weight;  root[i].data=root[i].weight;  root[i].LeftChild = NULL;  root[i].RightChild = NULL;  root[i].Parent = NULL;  }  //初始化非叶子节点  for(int i=m;i<2\*m-1;i++){  root[i].LeftChild = NULL;  root[i].RightChild = NULL;  root[i].Parent = NULL;  root[i].weight = 0;  }  //对叶子节点进行排序  for(int i=0;i<m-1;i++){  for(int j=i;j<m;j++){  if(root[i].weight>root[j].weight){  HTNode temp = root[i];  root[i] = root[j];  root[j] = temp;  }  }  }  //构造哈夫曼树  for(int i=0;i<m-1;i++){  int min1\_index = 0, min2\_index = 0;  int min1 = 1000000, min2 = 1000000;  for(int j=i;j<m+i;j++){  if(root[j].Parent==NULL){  if(root[j].weight<min1){  min2 = min1;  min2\_index = min1\_index;  min1 = root[j].weight;  min1\_index = j;  }else if(root[j].weight<min2){  min2 = root[j].weight;  min2\_index = j;  }  }  }  root[min1\_index].Parent = &root[m+i];  root[min2\_index].Parent = &root[m+i];  root[m+i].weight = root[min1\_index].weight + root[min2\_index].weight;  root[m+i].LeftChild = &root[min1\_index];  root[m+i].RightChild = &root[min2\_index];  }  root = &root[2\*m-2];  }  //创建二叉树  void createHafuTree\_Noinput(HuffTree &root,std::vector<cv::Point2d> pixels){  int m;  m = pixels.size();  root = new HTNode[2\*m-1];  //初始化叶子节点  int i=0;  for(auto pixel:pixels){  root[i].weight=pixel.y;  root[i].data=pixel.x;  root[i].LeftChild = NULL;  root[i].RightChild = NULL;  root[i].Parent = NULL;  i++;  }  //初始化非叶子节点  for(int i=m;i<2\*m-1;i++){  root[i].LeftChild = NULL;  root[i].RightChild = NULL;  root[i].Parent = NULL;  root[i].weight = 0;  root[i].data = 0;  }  //对叶子节点进行排序  for(int i=0;i<m-1;i++){  for(int j=i;j<m;j++){  if(root[i].weight>root[j].weight){  HTNode temp = root[i];  root[i] = root[j];  root[j] = temp;  }  }  }  //构造哈夫曼树  for(int i=0;i<m-1;i++){  int min1\_index = 0, min2\_index = 0;  int min1 = 1000000, min2 = 1000000;  for(int j=i;j<m+i;j++){  if(root[j].Parent==NULL){  if(root[j].weight<min1){  min2 = min1;  min2\_index = min1\_index;  min1 = root[j].weight;  min1\_index = j;  }else if(root[j].weight<min2){  min2 = root[j].weight;  min2\_index = j;  }  }  }  root[min1\_index].Parent = &root[m+i];  root[min2\_index].Parent = &root[m+i];  root[m+i].weight = root[min1\_index].weight + root[min2\_index].weight;  root[m+i].LeftChild = &root[min1\_index];  root[m+i].RightChild = &root[min2\_index];  }  root = &root[2\*m-2];  }  void printHafuTree(HuffTree root){  if(root){  printHafuTree(root->LeftChild);  printHafuTree(root->RightChild);  }else{  }  }  //生成哈夫曼编码  void generateHuffmanCodes(HuffTree root, std::string code, std::map<int, std::string>& codes) {  if (root->LeftChild) {  generateHuffmanCodes(root->LeftChild, code + "0", codes);  }  if (root->RightChild) {  generateHuffmanCodes(root->RightChild, code + "1", codes);  }  if (!root->LeftChild && !root->RightChild) {  codes[root->data] = code;  }  }  //画出哈夫曼树  void DrawHafuNode(HafuTree \*hafu,cv::Mat &img,cv::Point root,int linelength)  {  cv::Point left(root.x-linelength,root.y+100);  cv::Point right(root.x+linelength,root.y+100);  if(hafu){  cv::circle(img,root,16,cv::Scalar(255,0,0),1);  //填充根的数据  int i = hafu->weight;  cv::putText(img,std::to\_string(i),cv::Point(root.x-5,root.y+5),cv::FONT\_HERSHEY\_SIMPLEX,0.5,cv::Scalar(0,0,255),1.5);  //画出左右子树  if(hafu->LeftChild){  cv::line(img,cv::Point(root.x-13,root.y+13),cv::Point(left.x+13,left.y-13),cv::Scalar(0,255,0),2);  cv::putText(img,"0",cv::Point((root.x+left.x)/2.0,(root.y+left.y)/2.0-5),cv::FONT\_HERSHEY\_SIMPLEX,0.8,cv::Scalar(0,0,0),1);  DrawHafuNode(hafu->LeftChild,img,left,linelength-16);  }  if(hafu->RightChild){  cv::line(img,cv::Point(root.x+13,root.y+13),cv::Point(right.x-13,right.y-13),cv::Scalar(0,255,0),2);  cv::putText(img,"1",cv::Point((root.x+right.x)/2.0,(root.y+right.y)/2.0-5),cv::FONT\_HERSHEY\_SIMPLEX,0.8,cv::Scalar(0,0,0),1);  DrawHafuNode(hafu->RightChild,img,right,linelength-16);  }  }  }  //画出哈夫曼树  void DrawHafuTree(HafuTree \*T){  cv::Mat img(1000,1000,CV\_8UC3,cv::Scalar(255,255,255));  cv::Point root(500,50);  int linelength = 100;  DrawHafuNode(T,img,root,linelength);  std::map<int, std::string> huffmanCodes;  generateHuffmanCodes(T, "", huffmanCodes);  int show\_y=100;  cv::putText(img,"HuffmanCodes:",cv::Point(16,show\_y-16),cv::FONT\_HERSHEY\_SIMPLEX,0.5,cv::Scalar(18,0,255),1);  for (const auto& pair : huffmanCodes) {  std::cout << pair.first << ": " << pair.second << std::endl;  cv::putText(img,std::to\_string(pair.first)+":"+pair.second,cv::Point(16,show\_y),cv::FONT\_HERSHEY\_SIMPLEX,0.5,cv::Scalar(18,0,255),1);  show\_y+=16;  }  cv::imshow("hafuTree",img);  cv::waitKey(1000);  }  //编码哈夫曼树  void encodeHaffTree(HuffTree root, const std::string& inputstring, const std::string& path = "../encode.txt") {  std::map<int, std::string> huffmanCodes;  generateHuffmanCodes(root, "", huffmanCodes);  std::ofstream outputFile(path);  if (!outputFile.is\_open()) {  std::cerr << "无法打开文件" << std::endl;  return;  }  //  std::istringstream inputStream(inputstring);  std::vector<int> numbers;  int num =0;  while (inputStream >> num) {  numbers.push\_back(num);  }  //  for(int i=0;i<numbers.size();i++){  auto it = huffmanCodes.find(numbers[i]);  if (it != huffmanCodes.end()) {  outputFile << it->second;  } else {  std::cerr << "值 " << numbers[i] << " 没有对应的哈夫曼编码" << std::endl;  }  }  outputFile.close();  }  //译码哈夫曼树  void decodeHaffTree(HuffTree root,std::string path1="../encode.txt",std::string path2="../decode.txt") {  std::string inputFile = path1;  std::string outputFile = path2;  FILE \*input = fopen(inputFile.c\_str(), "r");  FILE \*output = fopen(outputFile.c\_str(), "w");  char bit;  HuffTree currentNode = root;  if (!input || !output) {  std::cerr << "无法打开文件" << std::endl;  return;  }  while (fscanf(input, "%c", &bit) != EOF) {  if (bit == '0') {  currentNode = currentNode->LeftChild;  }  else if (bit == '1') {  currentNode = currentNode->RightChild;  }  if (!currentNode->LeftChild && !currentNode->RightChild) {  fprintf(output,"%d ", currentNode->weight);  currentNode = root;  }  }  fclose(input);  }  //图像压缩  void imgCompression(std::string imgpath){  HafuTree \*root;  //为了方便实现，将图像转换为灰度图像  cv::Mat img = cv::imread(imgpath,cv::IMREAD\_GRAYSCALE);  // cv::cvtColor(img,img,cv::COLOR\_BGR2GRAY);  std::vector<cv::Point2d> pixels;  int realpixels[256]={0};  // 获取图像的行数和列数  int rows = img.rows;  int cols = img.cols;  // 遍历图像的每个像素  for (int i = 0; i < rows; ++i){  for (int j = 0; j < cols; ++j){  int pixel = img.at<uchar>(i, j);  realpixels[pixel]++;  }  }  //将像素值存入set中  for(int i=0;i<255;i++){  if(realpixels[i]!=0){  pixels.push\_back(cv::Point2d(i,realpixels[i]));  }  }  // 创建哈夫曼树  createHafuTree\_Noinput(root,pixels);  //生成哈弗曼编码  std::map<int, std::string> huffmanCodes;  generateHuffmanCodes(root, "", huffmanCodes);  //保存哈夫曼编码  std::string outputFile = imgpath+"\_huffmanCodes.txt";  std::ofstream output(outputFile);  for (const auto& pair : huffmanCodes) {  // Convert binary string to integer  int s = std::stoi(pair.second, nullptr, 2);  // Convert integer to 16-bit binary string  std::bitset<16> sss(s);  // Save the result to the output file  output << pair.first << ":" << sss.to\_string() << std::endl;  }  cv::Mat img\_compression = img;  // 遍历图像的每个像素，将像素值替换为哈夫曼编码  for(int i=0;i<rows;i++){  for(int j=0;j<cols;j++){  int pixel = img.at<uchar>(i,j);  std::string huff = huffmanCodes[pixel];  // 将哈夫曼编码字符串转换为 std::bitset  std::bitset<16> binaryNumber(huffmanCodes[pixel]); // 假设最大长度为32位  // 将二进制数转换为整数  int compressedValue = static\_cast<int>(binaryNumber.to\_ulong());  // 将整数值赋给 img\_compression 中的像素  img\_compression.at<uchar>(i, j) = static\_cast<uchar>(compressedValue);  }  }  //保存压缩后的图像  imwrite(imgpath+"\_compression.jpg",img\_compression);  }  void imgDeCompression(const std::string& imgCompressionpath, const std::string& huffmanCodespath) {  std::ifstream input(huffmanCodespath);  std::map< std::string,int> huffmanCodes;  //异常处理  if (!input.is\_open()) {  std::cerr << "Error opening Huffman codes file." << std::endl;  return;  }  std::string line;  //读取哈夫曼编码  while (std::getline(input, line)) {  std::istringstream iss(line);  int key;  std::string key\_1;  std::string value;  if (std::getline(iss, key\_1, ':') && std::getline(iss, value)) {  key = std::stoi(key\_1); // 将字符串转换为整数  huffmanCodes.insert({ value,key });  }  }  //读取压缩图像  cv::Mat img\_compression = cv::imread(imgCompressionpath,cv::IMREAD\_GRAYSCALE);  //转灰度图  //cv::cvtColor(img\_compression,img\_compression,cv::COLOR\_BGR2GRAY);  // 获取图像的行数和列数  int rows = img\_compression.rows;  int cols = img\_compression.cols;  // 创建一个空白图像  cv::Mat img\_decompression(rows, cols, CV\_8U, cv::Scalar(0));  for(int i=0;i<img\_compression.rows;i++){  for(int j=0;j<img\_compression.cols;j++){  int pixel = img\_compression.at<uchar>(i,j);  // img\_compression.at<uchar>(i, j) = huffmanCodes[pixel];  // 将像素值转换为二进制数  std::bitset<16> binaryNumber(pixel);  // 将二进制数转换为字符串  std::string binaryString = binaryNumber.to\_string();  // 从哈夫曼编码中查找对应的像素值  int haff= huffmanCodes[binaryString];  img\_decompression.at<uchar>(i, j) = haff;  }  }  cv::imwrite(imgCompressionpath+"\_decompression.jpg",img\_decompression);  input.close();  }  int main(){  HuffTree root;  createHafuTree(root);  printHafuTree(root);  DrawHafuTree(root);  std::map<int, std::string> huffmanCodes;  generateHuffmanCodes(root, "", huffmanCodes);  getchar();  std::string encodestring;  std::cout<<"输入编码数据,用空格隔开:";  //windows下请改成\r  std::getline(std::cin, encodestring);  encodeHaffTree(root,encodestring);  decodeHaffTree(root);  imgCompression("../运行截图.png");  imgDeCompression("../运行截图.png\_compression.jpg","../运行截图.png\_huffmanCodes.txt");  delete root;  }   1. **问题定义及需求分析**   成本问题  1.问题描述:  农夫J要修理围着牧场的长度很小一段栅栏。农夫J测量了栅栏，发现维修栅栏需要N块木头，木头的长度为整数Li个单位。他购买了一条较长的能锯开的木头，这里可忽略损耗，因为没有锯子，J向农夫D求助，D要求J锯N-1次每一次都要计算费用，并且支付的费用等于该段木头的长度，如长度为21的木头就要付21美分。例如，将长度为21的木头锯成长度为8、5、8三段。  方案1：第1次锯木头花费21美分，锯为13、8；第2次花费13，锯成8和5；总花费为21+13=34；  方案2：第一次第1次锯木头花费21美分，锯为16、5；第2次花费16，锯成8和5；总花费为21+16=37。  方案2比方案1花费高。D让J决定锯木头的次序和位置，请帮助农夫J确定锯木头的方案，使其花费较少的费用。  本题可抽象为哈夫曼编码问题（以下均在此基础上给出实验要求、实验思路和题目代码）。利用哈夫曼树求得用于通信的二进制编码称为哈夫曼编码。以N中字符出现的频率作为权值，设计电文总长度最短的二进制前缀编码（哈夫曼编码）。  2.实验要求:  （1）采用二叉树等存储结构。  （2）创建哈夫曼树，生成哈夫曼编码。  （3）编码文件的译码。  （4）可尝试位图文件的压缩问题。  3.程序的功能：  输出哈夫曼编码,压缩解压缩图片，用给定的节点构建哈夫曼树  4.测试数据以及结果 **:**  输入数据：一堆值，需要编码的字符串，一张图片  输出数据：哈夫曼树，哈夫曼编码，字符串的哈夫曼编码，图片  输出形式: 窗口打印，终端打印，以及文件  **测试样例1：**  **终端输出：**  请输入叶子节点的个数：4  请输入第1个叶子节点的权值：12  请输入第2个叶子节点的权值：13  请输入第3个叶子节点的权值：1  请输入第4个叶子节点的权值：4  1: 100  4: 101  12: 11  13: 0  输入编码数据,用空格隔开:12 13 1 4 4 1 113  值 113 没有对应的哈夫曼编码  **窗口输出：**    **文件输出：**  **decode.txt:**  **encode.txt:**  图片压缩哈夫曼码表：  3:0000100011011011  4:0000000000000001  22:0000000000010111  25:0000000000010010  30:0000000000011101  31:0000000000101011  37:0000000000111000  46:0000000010001100  53:0000000010100000  54:0000000011010110  55:0000000011010101  56:0000000010101000  58:0000000011010111  60:0000000010001110  62:0000000000100010  63:0000000000110111  66:0000000000010000  72:0000000000011001  73:0000000000100110  76:0000000000110001  78:0000000000100111  79:0000000001111000  80:0000000001110010  82:0000000001100001  84:0000000001011001  88:0000000001111011  93:0000000001100000  97:0000000000101101  99:0000000000101001  100:0000000000111110  101:0000000000110110  111:0000000000110100  112:0000000001010001  113:0000000001111001  118:0000000010110001  122:0000000100011110  126:0000000111101000  128:0000000101000011  130:0000001101010001  152:0000001110011101  153:0000001110011100  159:0000000111101001  160:0000001101010010  163:0000000111001111  169:0000000111101010  177:0000001000110111  187:0000000100011111  188:0000000100011010  199:0000011010100111  201:0000011010100001  205:0000110101001100  208:0000000010110000  210:0000010001101100  211:0000011010100000  215:0000000101000010  217:0000100011011010  219:0000110101001101  224:0000000010101001  228:0000000111101011  237:0000000001010101  242:0000000011100110  249:0000000000111111  250:0000000000000000  原图：    **压缩图：**    **解压缩图：**     1. **概要设计:**   **数据类型：**  HuffTree（哈夫曼树）：  代表哈夫曼树中的一个节点的结构。  包含权重、数据、左孩子、右孩子和父节点的字段。  Huffman树指针（HuffTree\*）：  指向哈夫曼树根的指针。  **主程序的流程如下：**  1.创建树：  用户输入叶子节点权重，或从图像中读取像素值。  构建哈夫曼树。  2.可视化：  使用OpenCV可视化哈夫曼树。  3.用户输入：  用户输入一系列数字进行编码。  4.编码：  使用哈夫曼编码对输入序列进行编码。  将编码结果写入文件。  5.解码：  使用哈夫曼树解码编码文件。  6.图像压缩和解压：  加载图像并使用哈夫曼编码进行压缩。  然后对压缩的图像进行解压。  程序模块之间的调用关系如下：  **3、 详细设计**  **以下是对概要设计中定义的数据类型、存储结构和每个模块的伪码算法的实现概要设计：**  **### 1数据类型和存储结构**  **##1. 数据类型 SetList 和 Setnode**  typedef struct HafuTree {  int weight;  float data;  HafuTree\* LeftChild;  HafuTree\* RightChild;  HafuTree\* Parent;  } \* HuffTree, HTNode;  这定义了一个简单哈夫曼树节点  **###2模块和操作的伪码算法**  **1. 创建哈夫曼树**  void createHafuTree(HuffTree& root) {  int m;  Input: 用户输入叶子节点个数 m  初始化 root 为大小为 2\*m-1 的节点数组  // 初始化叶子节点  for i in 0 to m-1 {  输入第 i+1 个叶子节点的权值  初始化 root[i] 的相关属性  }  // 初始化非叶子节点  for i in m to 2\*m-2 {  初始化 root[i] 的相关属性  }  // 对叶子节点按权值排序  对 root[0] 到 root[m-1] 按权值升序排序  // 构造哈夫曼树  for i in 0 to m-2 {  找到权值最小的两个节点 min1 和 min2  创建新节点 new\_node，以 min1 和 min2 为子节点，更新相关属性  }  root 指向构造好的哈夫曼树的根节点  }  **2. 创建哈夫曼树（无用户输入）**  void createHafuTree\_Noinput(HuffTree& root, std::vector<cv::Point2d> pixels) {  m = pixels.size()  初始化 root 为大小为 2\*m-1 的节点数组  // 初始化叶子节点  for i in 0 to m-1 {  使用 pixels[i] 初始化 root[i] 的相关属性  }  // 初始化非叶子节点  for i in m to 2\*m-2 {  初始化 root[i] 的相关属性  }  // 对叶子节点按权值排序  对 root[0] 到 root[m-1] 按权值升序排序  // 构造哈夫曼树  for i in 0 to m-2 {  找到权值最小的两个节点 min1 和 min2  创建新节点 new\_node，以 min1 和 min2 为子节点，更新相关属性  }  root 指向构造好的哈夫曼树的根节点  }  **3. 打印哈夫曼树**  void printHafuTree(HuffTree root) {  递归打印 root 的左子树  递归打印 root 的右子树  }  **4. 生成哈夫曼编码**  void generateHuffmanCodes(HuffTree root, std::string code, std::map<int, std::string>& codes) {  如果 root 有左子树，则递归生成左子树的编码（code + "0"）  如果 root 有右子树，则递归生成右子树的编码（code + "1"）  如果 root 为叶子节点，则将编码存入 codes 中  }  **5. 画出哈夫曼树**  void DrawHafuTree(HuffTree\* T) {  创建图像 img  创建根节点 root 在图像中的位置 root  递归画出哈夫曼树节点和连接线  显示图像  }  **6. 编码哈夫曼树**  void encodeHaffTree(HuffTree root, const std::string& inputstring, const std::string& path) {  生成哈夫曼编码表 huffmanCodes  打开文件 path 进行写入  从 inputstring 中读取每个数，查找其哈夫曼编码并写入文件  关闭文件  }  **7. 译码哈夫曼树**  void decodeHaffTree(HuffTree root, std::string path1, std::string path2) {  打开文件 path1 进行读取  打开文件 path2 进行写入  读取每个比特位，根据哈夫曼树进行译码，并将结果写入文件  关闭文件  }  **8. 图像压缩**  void imgCompression(std::string imgpath) {  创建哈夫曼树 root  将图像转换为灰度图像  统计图像像素及其频率，存储在 pixels 中  创建哈夫曼树（无用户输入）并生成哈夫曼编码表  保存哈夫曼编码表到文件  遍历图像每个像素，用哈夫曼编码替换像素值  保存压缩后的图像  }  **9. 图像解压**  void imgDeCompression(const std::string& imgCompressionpath, const std::string& huffmanCodespath) {  从文件读取哈夫曼编码表  读取压缩图像  遍历压缩图像每个像素，根据哈夫曼编码表进行解码  保存解压后的图像  }  **4、 调试分析**  问题**1**：内存泄漏：程序中使用了**new**分配内存，但在释放内存时需要确保没有内存泄漏。如果忘记释放分配的内存，程序可能会在长时间运行后耗尽内存。  改进：在结束程序时统一释放内存  问题**2**：输入字符串溢出缓冲区问题  改进：输入前清空缓存区  问题**3**：**算法的时空分析和改进设想:**  **时间复杂度：**  createHafuTree 和 createHafuTree\_Noinput 函数中的排序步骤可能会导致较高的时间复杂度，特别是在对叶子节点进行排序的时候。考虑使用更有效的排序算法，例如快速排序（std::sort）。  generateHuffmanCodes 函数中，遍历哈夫曼树并生成编码的过程的时间复杂度为 O(n)，其中 n 是哈夫曼树的节点数。  imgCompression 和 imgDeCompression 函数中，遍历图像的过程的时间复杂度为 O(rows \* cols)，其中 rows 和 cols 分别是图像的行数和列数。  **空间复杂度：**  createHafuTree 和 createHafuTree\_Noinput 函数中，为哈夫曼树节点分配了额外的空间，空间复杂度为 O(n)，其中 n 是叶子节点的个数。  generateHuffmanCodes 函数中使用了递归，可能导致堆栈空间的额外开销。  在 imgCompression 和 imgDeCompression 函数中，额外的空间用于存储哈夫曼编码和解码表，空间复杂度为 O(1)。  **改进设想：**  1.使用更有效的排序算法：  在排序叶子节点的步骤中，可以考虑使用更为高效的排序算法，例如快速排序，以减少排序的时间复杂度。   1. 非等长编码会导致无法解压图片   非等长编码存在无法保存到图片哈夫曼码表的问题，于是以16长度作为定长，更改编码  3.避免递归：  在 generateHuffmanCodes 函数中，递归生成哈夫曼编码。可以考虑使用迭代或非递归方法，以避免潜在的堆栈溢出问题。  4.优化哈夫曼编码查找：  在图像压缩和解压过程中，哈夫曼编码的查找可以通过使用哈希表或查找表来加速。这样可以减少对哈夫曼树的多次遍历。  **心得与体会：**  在编写程序之前，良好的概要设计和伪码算法是非常有帮助的。它们可以帮助你更好地理解问题，规划程序的结构，减少出错的机会。在开发期间，经常进行测试和调试是至关重要的。使用调试工具和打印语句来查找和修复问题，确保程序的正确性。在处理动态内存分配时，始终要确保释放分配的内存，以防止内存泄漏。在程序中添加足够的输入验证和错误处理机制，以提高程序的稳健性，防止不良输入导致程序崩溃。不断学习和改进。编程是一个不断学习和改进的过程。通过阅读相关文档、学习更高效的算法和数据结构，以及参考他人的经验，可以不断提高编程技能。  在代码编写过程中，考虑以下原则：  模块化设计： 将程序拆分为小模块，每个模块负责一个特定的功能。这有助于提高代码的可维护性和可读性。  合理的命名和注释： 使用有意义的变量和函数名，添加清晰的注释，使代码易于理解。这有助于你和其他人更容易阅读和理解代码。  测试驱动开发： 在编写代码之前先编写测试用例，确保代码在实现功能的同时保持正确性。  版本控制： 使用版本控制系统（如Git）来跟踪代码的变化，这样可以随时回滚到之前的版本，防止不小心引入的错误。  异常处理： 考虑程序可能面临的各种情况，并添加适当的异常处理机制，以确保程序在遇到问题时能够 graceful 地处理而不崩溃。  性能优化： 在需要的情况下进行性能分析，优化代码以提高程序的执行效率。  **5.使用程序：**  **预处理：**安装cmake（版本大于3.22），opencv和vscode，下载代码以及配置文件等,附下载地址:https://github.com/lzysbl/huffmanTree  **使用步骤**  **第一步：**  从键盘输入一个数字n作为节点中元素个数，回车，再依次输入n个数字作为节点权重同时也是节点数据，节点构成的哈夫曼树和编码会在hafuTree窗口显示出来，同时在窗口打印哈夫曼编码  **第二步：**  从键盘输入一些用空格隔开节点数值的字符创，作为需要编码的字符串，编码的字符串的编码会存入文件encode.txt,而解码的字符串会存入decode.txt文件  **第三步：**  输入图片路径（在本程序改为了默认，无需输入），程序对图片压缩，保存压缩图片，在解压得到解压图片，保存解压图片，保存哈夫曼码表。  **第四步：**  程序结束 | | | | | | |