编程实验：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 1. 实验目的与要求  在学习和理解二叉树的原理、构造及遍历方法的基础上，应用所学知识来解决实际问题。  本实验将通过一个实际应用问题的解决过程掌握Huffman树的构造、Huffman编码的生成及基于所获得的Huffman编码压缩文本文件。  涉及的知识点包括树的构造、遍历及C语言位运算和二进制文件。  2. 实验内容  **Huffman编码文件压缩**  【问题描述】  编写一程序采用Huffman编码对一个正文文件进行压缩。具体压缩方法如下：  1.    对正文文件中字符(换行字符'\n'除外，不统计)按出现次数（即频率）进行统计。  2.    依据字符频率生成相应的Huffman树（未出现的字符不生成）。  3.    依据Huffman树生成相应字符的Huffman编码。  4.    依据字符Huffman编码压缩文件（即将源文件字符按照其Huffman编码输出）。  说明：  1.    只对文件中出现的字符生成Huffman树，注意：一定不要处理\n，即不要为其生成Huffman编码。  2.    采用ASCII码值为0的字符作为压缩文件的结束符（即可将其出现次数设为1来参与编码）。  3.    在生成Huffman树前，初始在对字符频率权重进行（由小至大）排序时，频率相同的字符ASCII编码值小的在前；新生成的权重节点插入到有序权重序列中时，若出现相同权重，则将新生成的权重节点插入到原有相同权重节点之后（采用稳定排序）。  4.   在生成Huffman树时，权重节点在前的作为左孩子节点，权重节点在后的作为右孩子节点。  5.    遍历Huffman树生成字符Huffman码时，左边为0右边为1。  6.   源文件是文本文件，字符采用ASCII编码，每个字符占8个二进制位；而采用Huffman编码后，高频字符编码长度较短（小于8位），因此最后输出时需要使用C语言中的位运算将字符的Huffman码依次输出到每个字节中。    【输入形式】  对当前目录下文件input.txt进行压缩。  【输出形式】  将压缩后结果输出到文件output.txt中，同时将压缩结果用十六进制形式（printf("%x",...)）输出到屏幕上，以便检查和查看结果。  3. 实验准备  1．文件下载    从教学平台（judge.buaa.edu.cn）课程下载区下载文件lab\_tree2.rar，该文件中包括了本实验中用到的文件huffman2student.c和input.txt：  l  huffman2student.c：该文件给出本实验程序的框架，框架中部分内容未完成（见下面相关实验步骤），通过本实验补充完成缺失的代码，使得程序运行后得到相应要求的运行结果；  l  input.txt：为本实验的测试数据。       2. huffman2student.c文件中相关数据结构说明    结构类型说明：  struct tnode {     //Huffman树结构节点类型        char c;        int weight;        struct tnode \*left;        struct tnode \*right;        } ;    结构类型struct tnode用来定义Huffman树的节点，其中；  1）对于树的叶节点，成员c和weight用来存放字符及其出现次数；对于非叶节点来说，c值可不用考虑，weight的值满足Huffman树非叶节点生成条件，若p为当前Huffman树节点指针，则有：  p->weight = p->left->weight + p->right->weigth；  2）成员left和right分别为Huffman树节点左右子树节点指针。  全局变量说明：    int Ccount[128]={0};         struct tnode \*Root=NULL;  char HCode[128][MAXSIZE]={0};        int Step=0;  FILE \*Src, \*Obj;  整型数组Ccount存放每个字符的出现次数，如Ccount[‘a’]表示字符a的出现次数。  变量Root为所生成的Huffman树的根节点指针。  数组HCode用于存储字符的Huffman编码，如HCode['a']为字符a的Huffman编码，本实验中为字符串”1000”。  变量Step为实验步骤状态变量，其取值为1、2、3、4，分别对应实验步骤1、2、3、4。  变量Src、Obj为输入输出的文件指针，分别用于打开输入文件“input.txt”和输出文件“output.txt”。  4. 实验步骤  【步骤1】  1)        实验要求  在程序文件huffman2student.c中“//【实验步骤1】开始”和“ //【实验步骤1】结束”间编写相应代码，以实现函数statCount，统计文本文件input.txt中字符出现频率。  *//【实验步骤1】开始*  *void statCount()*  *{*      *}*  *//【实验步骤1】结束*  2)        实验说明  函数statCount用来统计输入文件（文件指针为全局变量Src）中字符的出现次数（频率），并将字符出现次数存入全局变量数组Ccount中，如Ccount[‘a’]存放字符a的出现次数。  注意：**在该函数中Ccount[0]一定要置为1，即Ccount[0]=1**。编码值为0（’\0’）的字符用来作为压缩文件的结束符。  3)        实验结果  函数print1()用来打印输出步骤1的结果，即输出数组Ccount中字符出现次数多于0的字符及次数，编码值为0的字符用NUL表示。完成【步骤1】编码后，本地编译并运行该程序，并在标准输入中输入1，程序运行正确时在屏幕上将输出如下结果：    形状  低可信度描述已自动生成     图1步骤1运行结果  在本地运行正确的情况下，将你所编写的程序文件中//【实验步骤1】开始”和“ //【实验步骤1】结束”间的代码拷贝粘贴到实验报告后所附代码【实验步骤1】下的框中，然后点击提交按钮，若得到如下运行结果（**测试数据1**评判结果为**完全正确**）：    表明**实验步骤1：通过，**否则**：不通过。**  【步骤2】  1)        实验要求  在程序文件huffman2student.c中的 “//【实验步骤2】开始”和“ //【实验步骤2】结束”间编写相应代码，实现函数createHTree，该函数生成一个根结点指针为Root的Huffman树。  *//【实验步骤2】开始*  *void createHTree( )*  *{*      *}*  *//【实验步骤2】结束*  2)        实验说明  在程序文件huffman2student.c中函数createHTree将根据每个字符的出现次数（字符出现次数存放在全局数组Ccount中，Ccount[i]表示ASCII码值为i的字符出现次数），按照Huffman树生成规则，生成一棵Huffman树。  **算法提示：**  *1．依据数组Ccout中出现次数不为0的（ 即Ccount[i]>0）项，构造出树林F={T0, T1, ¼, Tm}，初始时Ti(0≤i≤m)为只有一个根结构的树，且根结点(叶结点)的权值为相应字符的出现次数的二叉树（每棵树结点的类型为struct tnode，其成员c为字符，weight为树节点权值）：*  *for(i=0; i<128; i++)*  *if(Ccount[i]>0){*  *p = (struct tnode \*)malloc(sizeof(struct tnode));*  *p->c = i; p->weight = Ccount[i];*  *p->left = p->right = NULL;*  *add p into F;*  *}*  *2．对树林F中每棵树按其根结点的权值由小至大进行排序（排序时，当权值weight相同时，字符c小的排在前面），得到一个有序树林F*  *3．while 树个数>1 in F*  *a)*   *将F中T0和T1作为左、右子树合并成为一棵新的二叉树T’，并令T’->weight= T0->weight+ T1->wei*  *b)*   *删除T0和T1 from F，同时将T’加入F。要求加入T’后F仍然有序。若F中有树根结点权值与T’相同，则T’应加入到其后*  *4．Root = T0 （Root为Huffman树的根结点指针。循环结束时，F中只有一个T0）*  注：在实现函数createHTree时，在框中还可根据需要定义其它函数，例如：   |  | | --- | | void myfun()  {    …  }  void createHTree()  {    …    myfun();    …  } |     3)        实验结果  函数print2()用来打印输出步骤2的结果，即按前序遍历方式遍历步骤2所生成（由全局变量Root所指向的）Huffman树结点字符信息。输出时编码值为0的字符用NUL表示、空格符用SP表示、制表符用TAB表示、回车符用CR表示。完成【步骤2】编码后，本地编译并运行该程序，并在标准输入中输入2，程序运行正确时在屏幕上将输出如下结果：     图2 步骤2运行结果  在本地运行正确的情况下，将你在本地所编写的程序文件中//【实验步骤2】开始”和“ //【实验步骤2】结束”间的代码拷贝粘贴到实验报告后所附代码【实验步骤2】下的框中，然后点击提交按钮，若得到如下运行结果（**测试数据2**评判结果为**完全正确**）：    表明**实验步骤2：通过，**否则**：不通过。**  【步骤3】  1)        实验要求  在程序文件huffman2student.c中的 “//【实验步骤3】开始”和“ //【实验步骤3】结束”间编写相应代码，实现函数makeHCode，该函数依据【实验步骤3】中所产生的Huffman树为文本中出现的每个字符生成对应的Huffman编码。遍历Huffman树生成字符Huffman码时，左边为0右边为1。  *//【实验步骤3】开始*  *void makeHCode( )*  *{*      *}*  *//【实验步骤3】结束*  2)        实验说明  【步骤3】依据【步骤2】所生成的根结点为Root的Huffman树生为文本中出现的每个字符生成相应的Huffman编码。全局变量HCode定义如下：  char HCode[128][MAXSIZE];  HCode变量用来存放每个字符的Huffman编码串，如HCode[‘e’]存放的是字母e的Huffman编码串，在本实验中实际值将为字符串”011”。  **算法提示：**  可编写一个按前序遍历方法对根节点为Root的树进行遍历的递归函数，并在遍历过程中用一个字符串来记录遍历节点时从根节点到当前节点的路径（经过的边），经过左边时记录为’0’，经过右边时记录为’1’；当遍历节点为叶节点时，将对应路径串存放到相应的HCode数组中，即执行strcpy(HCode[p->c],路径串)。  注：在实现函数makeHCode时，在框中还可根据需要定义其它函数，如调用一个有类于前序遍历的递归函数来遍历Huffman树生成字符的Huffman编码：   |  | | --- | | void visitHTree()  {    …  }  void makeHCode()  {    …    visitHTree();    …  } |   3)        实验结果  函数print3()用来打印输出步骤3的结果，即输出步骤3所生成的存储在全局变量HCode中非空字符的Huffman编码串。完成【步骤3】编码后，本地编译并运行该程序，并在标准输入中输入3，在屏幕上将输出ASCII字符与其Huffman编码对应表，冒号左边为字符，右边为其对应的Huffman编码，其中NUL表示ASCII编码为0的字符，SP表示空格字符编码值为0的字符用，程序运行正确时在屏幕上将输出如下结果：    图3 步骤3运行结果  在本地运行正确的情况下，将你在本地所编写的程序文件中//【实验步骤3】开始”和“ //【实验步骤3】结束”间的代码拷贝粘贴到实验报告后所附代码【实验步骤3】下的框中，然后点击提交按钮，若得到如下运行结果（**测试数据3**评判结果为**完全正确**）：  图形用户界面, 应用程序  描述已自动生成  表明**实验步骤3：通过，**否则**：不通过。**  【步骤4】  1)        实验要求  在程序文件huffman2student.c函数中的 “//【实验步骤4】开始”和“ //【实验步骤4】结束”间编写相应代码，实现函数atoHZIP，该函数依据【实验步骤3】中所生成的字符ASCII码与Huffman编码对应表（存储在全局变量HCode中，如HCode[‘e’]存放的是字符e对应的Huffman编码串，在本实验中值为字符串”011”），将原文本文件（文件指针为Src）内容（ASCII字符）转换为Huffman编码文件输出到文件output.txt（文件指针为Obj）中，以实现文件压缩。同时将输出结果用十六进制形式（printf("%x",...)）输出到屏幕上，以便检查和查看结果。  *//【实验步骤4】开始*  *void atoHZIP( )*  *{*      *}*  *//【实验步骤4】结束*  2)        实验说明  Huffman压缩原理：在当前Windows、Linux操作系统下，文本文件通常以ASCII编码方式存储和显示。ASCII编码是定长编码，每个字符固定占一个字节（即8位），如字符’e’的ASCII编码为十进制101（十六进制65，二进制为01100101）。而Huffman编码属于可变长编码，本实验中其依据文本中字符的频率进行编码，频率高的字符的编码长度短（小于8位），而频率低的字符的编码长度长（可能多于8位），如在本实验中，字符’ ’（空格）的出现频率最高（出现65次），其Huffman编码为111（占3位，远小于一个字节的8位），其它出现频率较高的字符，如字符’e’的Huffman编码为011、字符’o’的Huffman编码为111；字符’x’出现频率低（出现1次），其Huffman编码为10011110（占8位，刚好一个字节）（注意，在其它问题中，字符最长Huffman编码可能会超过8位）。正是由于高频字符编码短，将使得Huffman编码文件（按位）总长度要小于ASCII文本文件，以实现压缩文件的目的。    然而，将普通ASCII文本文件转换为变长编码的文件不便之处在于C语言中输入/输出函数数据处理的最小单位是一个字节（如putchar()），无法直接将Huffman（不定长）编码字符输出，在输出时需要将不定长编码序列转换为定长序列，按字节输出。而对于不定长编码，频率高的字符其编码要比一个字节短（如本实验中字符’e’的Huffman编码为011，不够一个字节，还需要和其它字符一起组成一个字节输出），频率低的编码可能超过一个字节。如何将不定长编码字符序列转换成定长字符序列输出，一个简单方法是：    1）根据输入字符序列将其Huffman编码串连接成一个（由0、1字符组成的）串；  2）然后依次读取该串中字符，依次放入到一个字节的相应位上；  3）若放满一个字节（即8位），可输出该字节；剩余的字符开始放入到下一个字节中；  4）重复步骤2和3，直到串中所有字符处理完。    下面通过实例来说明：   原始文件input.txt中内容以“I will…”开始，依据所生成的Huffman码表，字母I对应的Huffman编码串为“0101111”，空格对应“111”，w对应“1001110”，i对应“01010”，l对应     “11001”。因此，将其转换后得到一个Huffman编码串“01011111111001110010101100111001…”，由于在C中，最小输出单位是字节（共8位），因此，要通过C语言的位操作符将每8个01字符串放进一个字节中，如第一个8字符串“01011111”中的每个0和1放入到一个字符中十六进制（即printf（”%x”,c）输出时，屏幕上将显示5f）（如下图所示）。下面程序段将Huffman编码串每8个字符串放入一个字节（字符变量hc）中：  图示  中度可信度描述已自动生成  *char hc;*  *…*  *for(i=0; s[i] != ‘\0’; i++) {*  *hc = (hc << 1) | (s[i]-'0');*  *if((i+1)%8 == 0) {*  *fputc(hc,obj);     //输出到目标（压缩）文件中*  *printf("%x",hc);   //按十六进制输出到屏幕上*  *}*  *}*  *…*  说明：  1．当遇到源文本文件输入结束时，应将输入结束符的Huffman码放到Huffman编码串最后，即将编码串HCode[0]放到Huffman编码串最后。  2．在处理完成所有Huffman编码串时（如上述算法结束时），处理源文本最后一个字符（文件结束符）Huffman编码串（其编码串为“01001010”）时，可能出现如下情况：其子串”010”位于前一个字节中输出，而子串“01010”位于另（最后）一个字节的右5位中，需要将这5位左移至左端的头，最后3位补0，然后再输出最后一个字节。  图示  描述已自动生成  注：在实现函数atoHZIP时，在框中还可根据需要定义其它函数或全局变量，如：   |  | | --- | | void myfun()  {    …  }  void atoHZIP()  {    …    myfun();    …  } |   1)        实验结果  函数print4()用来打印输出步骤4的结果，即根据输出步骤3所生成的存储在全局变量HCode中Huffman编码串，依次对源文本文件（input.txt）的ASCII字符转换为Huffman编码字符输出到文件output.txt中，同时按十六进行输出到屏幕上。完成【步骤4】编码后，本地编译并运行该程序，并在标准输入中输入4，在屏幕上将输出：    图4 步骤4运行结  说明：  从屏幕输出结果可以看出，由于采用了不定长的Huffman编码，且出现频率高的字符的编码长度短，压缩后，文件大小由原来的370字节变为200字节，文件压缩了45.95%。    在本地运行正确的情况下，将你在本地所编写的程序文件中//【实验步骤4】开始”和“ //【实验步骤4】结束”间的代码拷贝粘贴到教学平台实验报告后所附代码【实验步骤4】下的框中，然后点击提交按钮，若得到如下运行结果（**测试数据4**评判结果为**完全正确**）：  文本  描述已自动生成  表明**实验步骤4：通过，**否则**：不通过。**  **\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_** | 20.0 |