哈夫曼编码设计文档

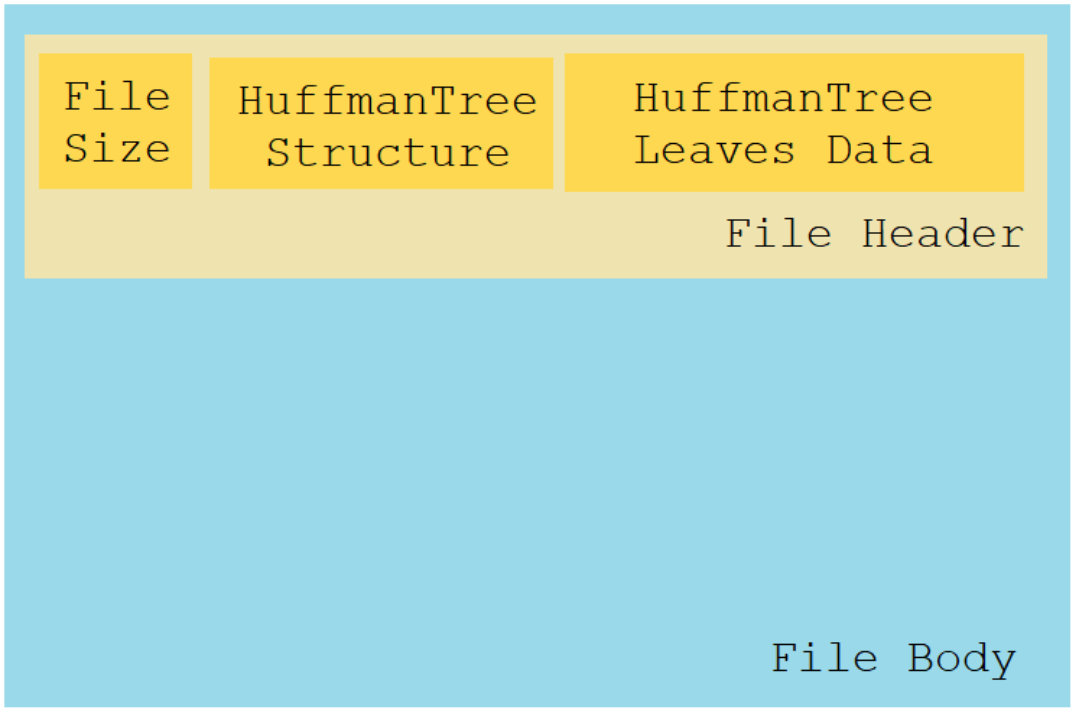
2021 年 12 月 1 日星期三

为了保证文件的可协程，我们必须采纳一个通用的**编码协议**。

以下部分是(JXF, LZC, YYH)小组的编码设计，如果您希望与我们进行通讯，您需要按照下面的思路进行编码设计。

如果您发现该设计有任何问题或者疑问，请直接联系我们!

● 概览



对一个文件进行哈夫曼编码，得到的文件将存在上图所示的结构。

哈夫曼编码算法将会对$[0000\ 0000]_2 \rightarrow [1111\ 1111]_2$共 256 个字节赋予权值，以此为基础建立一颗哈夫曼二叉树。编码之后，每个叶子节点内存贮编码前的字节，叶子节点的路径序列则为该字节的哈夫曼编码。

1. **File Body**

   文件的主体部分。即对源文件每个字节编码，其结果依次存放在 Body 部分。

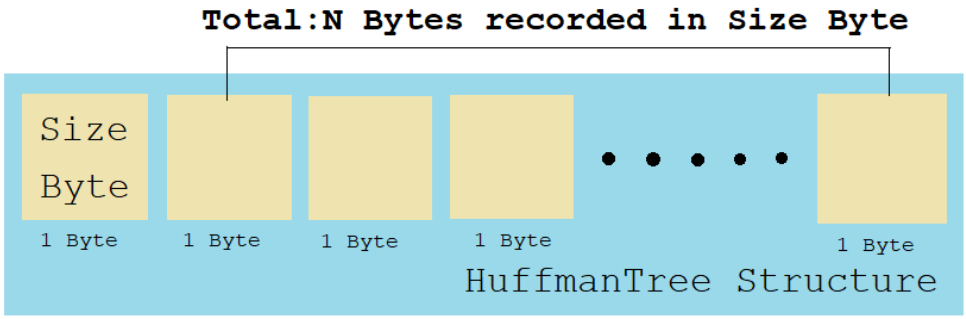
2. **File Header**

   a) **File Size**

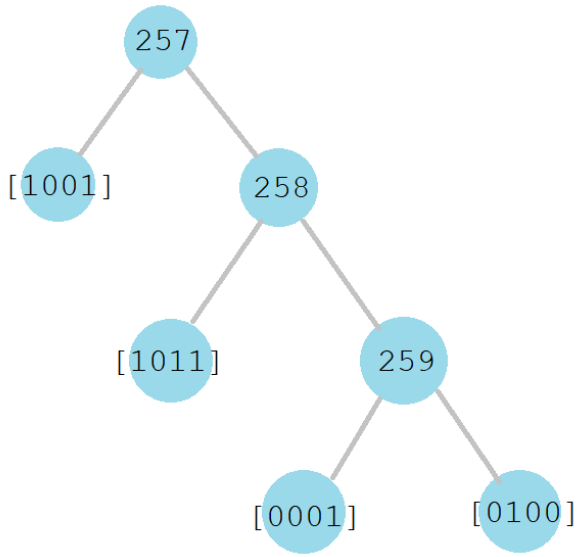
      存储文件大小，可以有（……）等作用

   b) **HuffmanTree Structure**

      这部分可分为两部分，首字节表示了哈夫曼树结构占用的字节数，剩余部分为表述树结构的 01 串。



对于一串编码后的文件，如果没有其对应的哈夫曼树，我们显然没法将其解码。因此，我们必须记录下这颗哈夫曼树。如何记录更节省空间呢？这里采用**双亲孩子存储法。**



假如我们根据源文件的出现字节的权值建立起上图所示的哈夫曼树。这是一颗简化版本的哈夫曼树，其中 257、258、259 为临时节点的标号，是没有意义的，不需要考虑。

1. **首先对这棵树进行先序遍历：**

$$257 \rightarrow [1001] \rightarrow 258 \rightarrow [1011] \rightarrow 259 \rightarrow [0001] \rightarrow [0100]$$

2. **每一次遍历到一个节点，需要记录下当前节点是否有左右孩子。我们用 1 表示存在，0 表示不存在。**

当前节点	257	[1001]	258	[1011]	259	[0001]	[0100]
是否有左孩子	1	0	1	0	1	0	0
是否有右孩子	1	0	1	0	1	0	0

按照左先右后的顺序，我们可以得到如下的 01 串

11 00 11 00 11 00 00

由于哈夫曼树的性质，**一个树节点要么存在两个孩子，要么都不存在**。这一点可以从构造的角度证明得到。故而我们可以将 01 串进一步简化为

1 0 1 0 1 0 0

如上 01 串即记录了一颗完整哈夫曼树的结构，不足 8 位则用 0 补全到 8 位。读者可以自行模拟这个过程以加深理解。

实际上，如果哈夫曼树比较大，使用了多个字节，但后面几个字节全都为 0，那么就可以把这些 0 以字节为单位全部舍去。例如：

[00010000] …… 00101000 00000000 00000000

后面两个字节是完全不需要记录的，我们只要把记录结构大小的字节-2 即可。但是考虑到某些现实因素，Default 程序暂时没有使用。

因为临时节点不保存数据，接下来部分只要按照先序遍历的顺

序**依次**存储叶子节点的数据即可。同时还需要记录下哈夫曼树结构 01 串所需要的字节大小。

上图哈夫曼树编码后的最终结果为

$$Size\ Byte: 0000\ 0001$$

$$Structure: 1010\ 1000$$

$$Data: 0000\ 1001$$

$$0000\ 1011$$

$$0000\ 0001$$

$$0000\ 0100$$

保存在文件中为:

$$00000001\ 10101000\ 00001001\ 00001011\ 00000001\ 00000100$$

c) HuffmanTree Leaves Data

按照先序遍历存储字节叶子的数据，即为上一部分中的 Data 部分。

现在给出编码/解码这颗哈夫曼树的 C++风格示例:

```
//对哈夫曼树进行编码
bool enCodingTree(FILE *fp , HuffmanTree ) {
    searchTree(HuffmanTree , top);

    //while(!buff[buff_t--]);
    fwrite(&(++buff_t) , 1 , 1 ,fp); //写入首字节:树结构数组的大小
    fwrite(buff , buff_t , 1 ,fp);    //写入 buff 数组:存储树结构的
    fwrite(bitSeq , bitSeq_p , 1 ,fp);//写入 bitseq 数组:按照 DFS 序,依次存储权值从大到小的字节叶子
    return true;
};
```

```
//递归的 DFS 哈夫曼树,进行编码
void searchTree(HuffmanTree , now) {
    bool Cflag = if(Node[now] has Children);   //判断是否有孩子

    buff[buff_t] = write bit[buff_p] then buff_p++;
```

```
    if(buff_p == 8) buff_t++ , buff_p = 0;   //当写满一个字节后,buff_t+1,存储到下一个 buff
    if(Cflag){ bitSeq[bitSeq_p++] = Value[now]; return ;} //如果是字节叶子,则存储当前节点的字节并返回

    searchTree( HuffmanTree , HuffmanTree[_left]); //遍历左子树
    searchTree( HuffmanTree , HuffmanTree[_right]); //遍历右子树
}
```

```
//从文件中读入头部分,进行解码并建树.返回树根的编号(默认为 256)
int deCodingTree(FILE *fp , HuffmanTree ) {
    fread(&head_t, 1 , 1 ,fp); //读入首字节:树结构数组的大小
    fread(buff, 1 , head_t , fp); //读入树结构数组.此后 fp 位于首个字节叶子

    int top = TOP_NUM;  buff_p = 0;
    buildTree(&fp , HuffmanTree , top);

    return TOP_NUM;
}
```

```
//DFS 遍历,建立哈夫曼树.返回当前节点是否为字节叶子
bool buildTree(FILE *fp , HuffmanTree , int &now){
    bool Cflag = if(buff[now] has Children); //从 buff 字符串的判断 now 节点是否有孩子

    if(!Cflag) {now--; return true;}     //当前是字节叶子节点.总编号-1 返回

    HuffmanTree[_left] = ++now;//首先尝试赋予一个新的临时编号
    if( buildTree(fp , HuffmanTree , now )) { //如果是字节叶子节点,则顺序读入一个字节叶子数据
        fread(&t , 1 , 1 , *fp);
        HuffmanTree[_left] = t;
    }
    HuffmanTree[_right] = ++now;
    if( buildTree(fp , HuffmanTree , now )) {
        fread(&t , 1 , 1 , *fp);
        HuffmanTree[_right] = t;
    }
    return false;
};
```