

哈夫曼压缩总结

C++实现



孙浩然

2020-4-16

吉林师范大学

目录

[一．引言 1](#_Toc38108275)

[二．主要技术点 1](#_Toc38108276)

[三．过程介绍 1](#_Toc38108277)

[1、压缩： 1](#_Toc38108278)

[2、解压缩 1](#_Toc38108279)

[四．详细分析 2](#_Toc38108280)

[1.准备过程 2](#_Toc38108281)

[2.建立哈夫曼树与哈夫曼编码 2](#_Toc38108282)

[哈夫曼树 3](#_Toc38108283)

[哈夫曼编码的概念 4](#_Toc38108284)

[3.哈夫曼树的实现 4](#_Toc38108285)

[4.哈夫曼编码压缩 7](#_Toc38108286)

[5.哈夫曼编码解压 11](#_Toc38108287)

# 一．引言

hffman编码的思想对文件进行压缩，主要原理是通过huffman编码来重新表示字符，使得出现频率高的字符编码短，出现少的字符编码长。整体下来的话，所需的总的bit位是减少的。但是要注意当大部分字符出现的频率都差不多时，huffman压缩的压缩效率会很低。

编程环境：Vscode

# 二．主要技术点

1. 文件的各种操作
2. 递归思想构造哈夫曼编码。
3. 位运算思想进行压缩与解压缩。
4. 自定义目标压缩文件头部元数据。

# 三．过程介绍

## 1．压缩：

先读取所有的字符，统计他们的频率，用他们的频率建立一个哈夫曼树，对哈夫曼树进行哈夫曼编码，再通过哈夫曼编码重新对源文件进行编码。并压入识别头文件和字符和其对应的频率表。

## 2．解压缩

对于压缩的过程的逆过程，需要decode编码Map，哈夫曼编码所对应的字符Map。

# 四．详细分析

## 1.准备过程

1、取出index位，若取出的index位为0，则GET\_BYTE值为假，否则为真

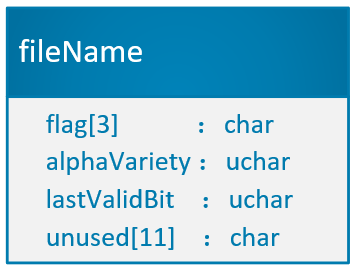
#define GET\_BYTE(vbyte, index) (((vbyte) & (1 << ((index) ^ 7))) != 0)

2、把index位设置为‘1’

#define SET\_BYTE(vbyte, index) ((vbyte) |= (1 << ((index) ^ 7)))

3、把index位设置为‘0’

#define CLR\_BYTE(vbyte, index) ((vbyte) &= (~(1 << ((index) ^ 7))))



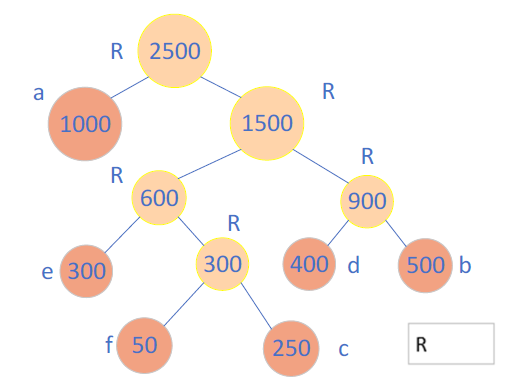


## 2.建立哈夫曼树与哈夫曼编码

### 哈夫曼树

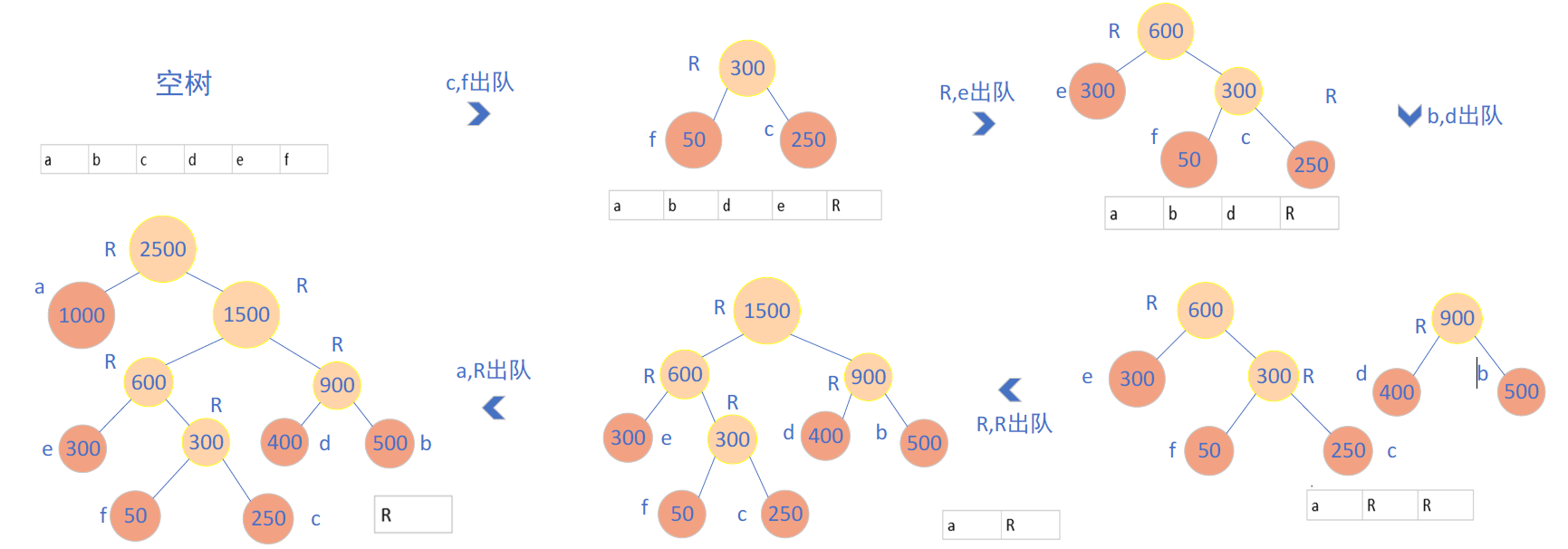
根据贪心算法的思想实现，把字符出现频率较多的字符用稍微短一点的编码，而出现频率较少的字符用稍微长一点的编码。哈夫曼树就是按照这种思想实现，下面将举例分析创建哈夫曼树的具体过程。下面表格的每一行分别对应字符及出现频率，根据这些信息就能创建一棵哈夫曼树。





哈夫曼编码之后的文件总二进制位为6300，而原本的一个字符八个二进制位，2500\*8=20000二进制位

如下图，将每个字符看作一个节点，将带有频率的字符全部放到优先队列中，每次从队列中取频率最小的两个节点 a 和 b（这里频率最小的 a 作为左子树），然后新建一个节点R，把节点设置为两个节点的频率之和，然后把这个新节点R作为节点A和B的父亲节点。最后再把R放到优先队列中。重复这个过程，直到队列中只有一个元素，即为哈夫曼树的根节点。（该优先队列也称为最小堆）



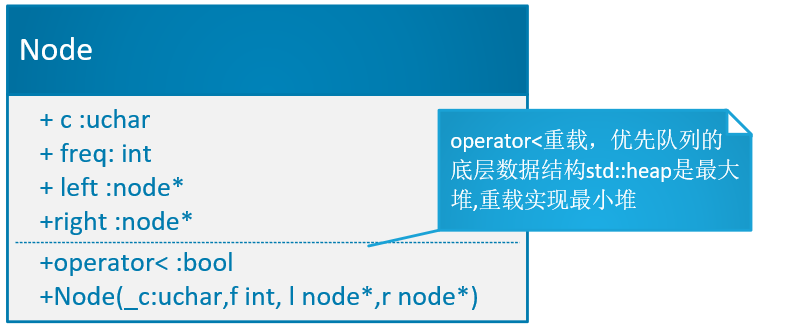
### 哈夫曼编码的概念

哈夫曼树又称作最优树，是一种带权路径长度最短的树，而通过哈夫曼树构造出的编码方式称作哈夫曼编码。

也就是说哈夫曼编码是一个通过哈夫曼树进行的一种编码，一般情况下，以字符 “0” 与 “1” 表示。编码的实现过程很简单，只要实现哈夫曼树，通过遍历哈夫曼树，这里我们从根节点开始向下遍历，如果下个节点是左孩子，则在字符串后面追加 “0”，如果为其右孩子，则在字符串后追加 “1”。结束条件为当前节点为叶子节点，得到的字符串就是叶子节点对应的字符的编码。

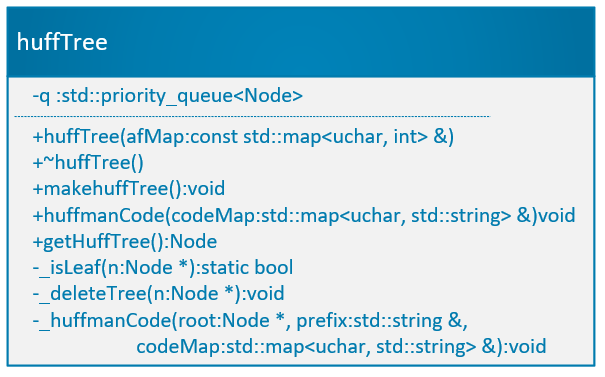
## 3.哈夫曼树的实现

哈夫曼树的结点：

  
首先定义哈弗曼树的结点，每个哈夫曼结点包括一个c记录当前字符。Freq记录每个字符出现的频率。

‘同时有两个子树结点指针。对哈夫曼结点重载运算符<，为了STL中使用最小堆建立哈夫曼树。

哈夫曼的UML



为了方便理解把上面UML图的函数分为几个部分解释

前提准备部分

|  |
| --- |
| Node getHuffTree() //获取哈夫曼树      {          return q.top();      }  static bool \_isLeaf(Node \*n) //判断是不是叶子结点      {          return n->left == nullptr && n->right == nullptr;      } |

判断是不是叶子结点设置为静态成员变量，可以节省空间上的开销

第一部分，建立哈夫曼树

afMap字符和其对应的出现频率

讲所有的字符和对应频率转换为哈夫曼的结点并进入优先队列q

之后构造哈夫曼树，构造哈夫曼树通过优先队列出队俩最小的（最小堆特点）合并并会到队列中。继续操作直到最后只有一个哈夫曼树。

|  |
| --- |
| huffTree(const std::map<uchar, int> &afMap)  {      for (auto i : afMap) //aotu自动遍历      {              Node n(i.first, i.second); //只有节点的树              q.push(n);                 //压入最小堆         }                              //构造森林          makehuffTree();                //合并树。最后合并为哈夫曼树  }  void makehuffTree()  {          while (q.size() != 1) //最后只剩一个哈夫曼树          {              Node \*left = new Node(q.top());              q.pop();              Node \*right = new Node(q.top());              q.pop();              Node node('R', left->freq + right->freq, left, right);              q.push(node);          }  } |

第二部分

递归删除哈夫曼树

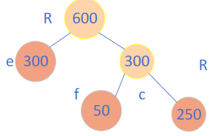
|  |
| --- |
| ~huffTree() //删除哈夫曼树      {          Node node = q.top();          \_deleteTree(node.left);          \_deleteTree(node.right);      }  void \_deleteTree(Node \*n) //递归删除树      {          if (!n)              return;          \_deleteTree(n->left);          \_deleteTree(n->right);          delete n;      } |

第三部分哈夫曼编码

获取哈夫曼树的头结点。

之后进行遍历，往左子树走前缀码设置一位0，往右子树走前缀码设置一位1。

直到叶子结点，然后确定每一个字符的前缀码。



例如这棵哈夫曼树。

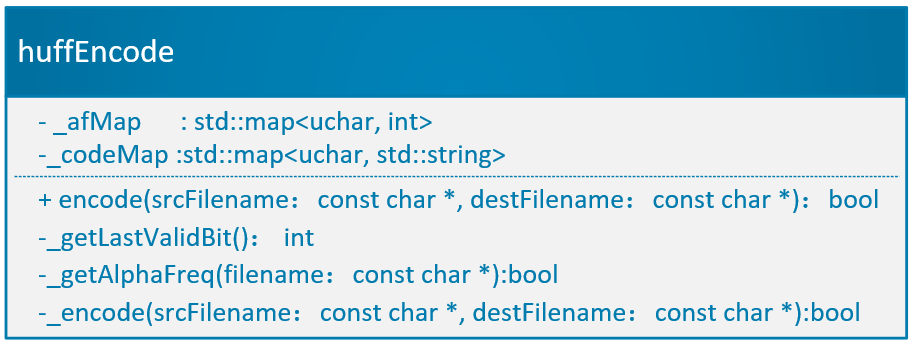
e的前缀码就是0

f的前缀码就是10

c的前缀码就是11

|  |
| --- |
| void huffmanCode(std::map<uchar, std::string> &codeMap)      {          Node node = q.top();//获取头结点          std::string prefix;          \_huffmanCode(&node, prefix, codeMap);      }  void \_huffmanCode(Node \*root, std::string &prefix,                        std::map<uchar, std::string> &codeMap)      {          std::string tmp = prefix;          if (root->left != nullptr)          {              prefix += '0';              if (\_isLeaf(root->left))              {                  codeMap[root->left->c] = prefix;              }              else              {                  \_huffmanCode(root->left, prefix, codeMap);              }          }          if (root->right != nullptr)          {              prefix = tmp;              prefix += '1';              if (\_isLeaf(root->right))              {                  codeMap[root->right->c] = prefix;              }              else              {                  \_huffmanCode(root->right, prefix, codeMap);              }          }      } |

## 4.哈夫曼编码压缩



\_afMap为字符及其对用的出现频率

\_codeMap为字符说对应的哈夫曼前缀码

encode即为程序员调用的一个函数

\_getLastValidBit()是最后一个字节可能有效位数不够8位，会产生垃圾位。所以判断一下有效位数

\_getAlphaFreq()是通过读取源文件获取对应的字符以及出现频率

\_encode为通过哈夫曼前缀码重新编码源文件并且写入新文件内

encode

|  |
| --- |
| bool encode(const char \*srcFilename, const char \*destFilename) //编码      {          if (!\_getAlphaFreq(srcFilename)) //如果没有找到文件返回false              return false;          huffTree htree(\_afMap); //用字符和出现频率创建一个哈夫曼树（创建细节请见hufftree.h）          htree.huffmanCode(\_codeMap); //通过哈夫曼树确定哈夫曼编码(编码细节请见hufftree.h)          return \_encode(srcFilename, destFilename); //将哈夫曼编码写入文件      } |

\_getLastValidBit()

|  |
| --- |
| int \_getLastValidBit() //获取最后的有效位数      {          int sum = 0;          for (auto it : \_codeMap)          {              sum += it.second.size() \* \_afMap.at(it.first);              sum &= 0xFF; //如果不执行这一步，当数据长度超过int的表示范围，就会出错              //0xFF化为二进制位1111 1111，这样做sum始终是最后一个字节,8位          }          //举例：若最后生成7位二进制，划分为一个字节，那么，这一个字节只有7位为有效位，其余都是垃圾位。          //我们只需要取出这个字节的那7个有效位，所以sum和8取余即可          //sum = sum % 8 <=> sum = sum & 0x7          //返回最后一个字节的有效位数          sum &= 0x7;          return sum == 0 ? 8 : sum;      } |

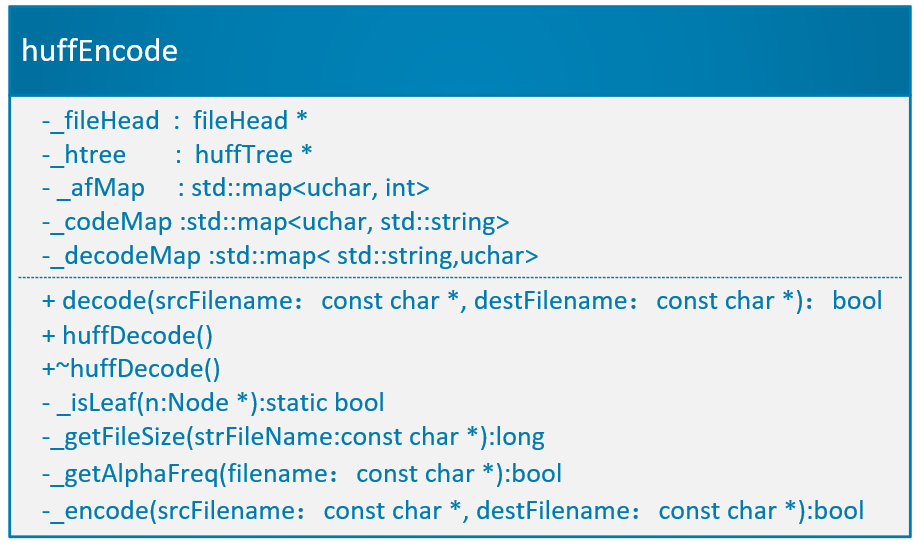
\_getAlphaFreq()

|  |
| --- |
| bool \_getAlphaFreq(const char \*filename)      {          uchar ch;          std::ifstream input(filename, std::ios::binary);          if (!input.is\_open())          {              printf("read file failed! filename: %s", filename);              return false;          }          //先读入一个字符，防止下面操作出错          input.read((char \*)&ch, sizeof(uchar));          while (!input.eof())          {              \_afMap[ch]++; //每个字符对应的频率加一              //读取下一个字符              input.read((char \*)&ch, sizeof(uchar));          }          input.close();          return true;      } |

\_encode

|  |
| --- |
| bool \_encode(const char \*srcFilename, const char \*destFilename)      {          uchar ch;                                     //字符          uchar value;                                  //写入用的临时数值          int bitIndex = 0;                             //位数统计          fileHead filehead = {'S', 'h', 'r'};          //识别文件的头          filehead.alphaVariety = (uchar)\_afMap.size(); //字符的种类          filehead.lastValidBit = \_getLastValidBit();   //最后一位有效数位          std::ifstream input(srcFilename, std::ios::binary);          if (!input.is\_open())          {              printf("read file failed! filename: %s", srcFilename);              return false;          }          std::ofstream output(destFilename, std::ios::binary);          if (!output.is\_open())          {              printf("read file failed! filename: %s", destFilename);              return false;          }          //写入识别文件          output.write((char \*)&filehead, sizeof(fileHead));          for (auto i : \_afMap)          {              alphaFreq af(i);                              //使用alphaFreq构造函数构造af              output.write((char \*)&af, sizeof(alphaFreq)); //字符和频率写入          }          //先读取一个字符，防止下面操作出错          input.read((char \*)&ch, sizeof(uchar));          while (!input.eof())          {              std::string code = \_codeMap.at(ch); //code为每一个huffmancode的编码值              for (auto c : code)                 //遍历每一位0或1              {                  //设置value的每个字节的位数，通过value写入1个字节1个字节写入                  if ('0' == c)                  {                      CLR\_BYTE(value, bitIndex);                  }                  else                  {                      SET\_BYTE(value, bitIndex);                  }                  ++bitIndex;                  if (bitIndex >= 8)                  {                      bitIndex = 0;                      output.write((char \*)&value, sizeof(uchar)); //一个字节一个写入                  }                  //这样写入最后一组数据前的编码（最后一组可能不够8位，会产生其他垃圾数据，对最后一组单独处理）              }              input.read((char \*)&ch, sizeof(uchar));          }          //写入最后一组的编码          if (bitIndex)          {              output.write((char \*)&value, sizeof(uchar));          }          input.close();          output.close();          return true;      } |

## 5.哈夫曼编码解压



\_afMap为字符及其对用的出现频率

\_codeMap为字符说对应的哈夫曼前缀码

\_decodeMap为哈夫曼前缀码对应字符

decode即为程序员调用的一个函数

\_getAlphaFreq为获取字符和其频率信息

\_decode对文件内容进行还原

\_GetFileSize（）

|  |
| --- |
| long \_getFileSize(const char \*strFileName) //获取文件大小      {          std::ifstream input(strFileName);          if (!input.is\_open())              return 0;          input.seekg(0, std::ios\_base::end);          std::streampos sp = input.tellg();          input.close();          return sp;      } |

\_getAlphaFreq

|  |
| --- |
| bool \_getAlphaFreq(const char \*filename)      {          std::ifstream is(filename, std::ios::binary);          if (!is)          {              printf("read file failed! filename: %s", filename);              return false;          }          //读取文件识别的文件          is.read((char \*)\_fileHead, sizeof(fileHead));          if (!(\_fileHead->flag[0] == 'S' &&                \_fileHead->flag[1] == 'h' &&                \_fileHead->flag[2] == 'r'))          {              printf("not support this file format! filename: %s\n", filename);              return false;          }          //根据字符种类按照alphaFreq大小读取          for (int i = 0; i < static\_cast<int>(\_fileHead->alphaVariety); ++i)          {              alphaFreq af;              is.read((char \*)&af, sizeof(af));              \_afMap.insert(std::pair<char, int>(af.alpha, af.freq)); //写入操作的逆过程          }          is.close();          return true;      } |

\_decode

|  |
| --- |
| bool \_decode(const char \*srcFilename,                   const char \*destFilename)      {          long fileSize = \_getFileSize(srcFilename);          std::ifstream is(srcFilename, std::ios::binary);          if (!is)          {              printf("read file failed! filename: %s", srcFilename);              return false;          }          //文件指针跳过头文件和写入的字符和其频率，读取哈夫曼编码,创建哈夫曼树还原源码          is.seekg(sizeof(fileHead) + sizeof(alphaFreq) \* \_fileHead->alphaVariety);          Node node = \_htree->getHuffTree(); //获取哈夫曼的根节点          Node \*pNode = &node;          std::ofstream io(destFilename, std::ios::binary);          if (!io)          {              printf("create file failed! filename: %s", destFilename);              return false;          }          uchar value;          std::string code;          int index = 0;          long curLocation = is.tellg(); //获取当前文件指针的位置          is.read((char \*)&value, sizeof(uchar));          while (true)          {              if (\_isLeaf(pNode))              {                  uchar alpha = \_decodeMap[code];                  io.write((char \*)&alpha, sizeof(uchar));                  if (curLocation >= fileSize && index >= \_fileHead->lastValidBit) //读取完所有数据，结束                  {                      break;                  }                  code.clear(); //清空数组                  pNode = &node;              }              if (GET\_BYTE(value, index))              {                  code += '1';                  pNode = pNode->right;              }              else              {                  pNode = pNode->left;                  code += '0';              }              if (++index >= 8)              {                  index = 0;                  is.read((char \*)&value, sizeof(uchar));                  curLocation = is.tellg();              }          }          is.close();          io.close();          return true;      } |

decode

|  |
| --- |
| bool decode(const char \*srcFilename, const char \*destFilename)      {          if (!\_getAlphaFreq(srcFilename))              return false;          long fileSize = \_getFileSize(srcFilename);          \_htree = new huffTree(\_afMap);          \_htree->huffmanCode(\_codeMap);          for (auto it : \_codeMap)          {              \_decodeMap.insert(std::pair<std::string, uchar>(it.second, it.first));          }          return \_decode(srcFilename, destFilename);      } |

# 五．GitHub代码