**东北大学计算机科学与工程学院**

**数据结构课程设计报告**

题目 哈夫曼压缩软件设计

课题组长 白千一

课题组成员 金忠信

专业名称 计算机科学与技术

班级 计1907

指导教师 杨雷

2021 年 1月

**课程设计任务书**

|  |
| --- |
| **题目：**  哈夫曼压缩软件设计 |
| **问题描述：**  采用哈夫曼树求得的用于通信的二进制编码称为哈夫曼编码。利用哈夫曼编码的数据压缩技术，设计文本格式的压缩软件或位图格式的压缩软件。 |
| **设计要求：**  设计基于哈夫曼编码的压缩软件。  （1）采用静态链表的二叉树等数据结构的类实现。  （2）创建哈夫曼树。  （3）哈夫曼编码和译码。  （4）源码、编码和压缩后的信息均以文件形式保存。  （5）软件时间和空间性能分析。  （6）基于哈夫曼编码的位图压缩软件设计（可选）。 |
| **指导教师签字：**  **年　　月　　日** |

目录

[1、 课题概述 - 1 -](#_Toc61896613)

[1.1课题任务 - 1 -](#_Toc61896614)

[1.2 课题原理 - 1 -](#_Toc61896615)

[1.3相关知识 - 1 -](#_Toc61896616)

[2、 需求分析 - 1 -](#_Toc61896617)

[2.1 课题调研 - 1 -](#_Toc61896618)

[2.2 用户需求分析 - 2 -](#_Toc61896619)

[3、 方案设计 - 2 -](#_Toc61896620)

[3.1总体功能设计 - 2 -](#_Toc61896621)

[3.2数据结构设计 - 3 -](#_Toc61896622)

[3.3类原型设计 - 3 -](#_Toc61896623)

[3.4主算法设计 - 8 -](#_Toc61896624)

[3.5用户界面设计 - 15 -](#_Toc61896625)

[4、 方案实现 - 17 -](#_Toc61896626)

[4.1开发环境与工具 - 17 -](#_Toc61896627)

[4.2程序设计关键技术 - 17 -](#_Toc61896628)

[4.3个人设计实现 - 17 -](#_Toc61896629)

[4.3.1白千一个人设计实现 - 17 -](#_Toc61896630)

[4.3.2金忠信个人设计实现 - 22 -](#_Toc61896631)

[5、 测试和调试 - 25 -](#_Toc61896632)

[5.1个人测试 - 26 -](#_Toc61896633)

[5.1.1白千一测试 - 26 -](#_Toc61896634)

[5.1.2金忠信测试 - 32 -](#_Toc61896635)

[5.2组装与系统测试 - 35 -](#_Toc61896636)

[6、 课题总结 - 42 -](#_Toc61896637)

[6.1课题评价 - 42 -](#_Toc61896638)

[6.2团队协作 - 42 -](#_Toc61896639)

[6.3下一步工作 - 42 -](#_Toc61896640)

[6.4个人设计小结 - 43 -](#_Toc61896641)

[6.4.1白千一设计小结 - 43 -](#_Toc61896642)

[6.4.2金忠信设计小结 - 43 -](#_Toc61896643)

[7、 附录 - 44 -](#_Toc61896644)

[A课题任务分工 - 44 -](#_Toc61896645)

[A-1课题程序设计分工 - 44 -](#_Toc61896646)

[A-2课题报告分工 - 45 -](#_Toc61896647)

[B课题设计文档 - 45 -](#_Toc61896648)

[B-1源程序代码（见附录项目文件夹） - 45 -](#_Toc61896649)

[B-2工程与可执行文件（见附录项目文件夹） - 45 -](#_Toc61896650)

1. 课题概述

## 1.1课题任务

利用哈夫曼编码的数据压缩技术，设计压缩软件。使用静态链表的二叉树等数据结构。实现能够将原码文件转化成编码文件或压缩文件，并能够将其复原为原码文件的功能。实现能够进行对文本格式和位图格式或其他合适的文件进行压缩的功能。在此基础上，拥有较好的时间和空间性能。

## 1.2 课题原理

1. 哈夫曼树

哈夫曼树又称最优二叉树，是一种带权路径长度最短的二叉树。树的带权路径长度，即树中所有的叶子结点的权值乘上其到根结点的路径长度（若根结点为 0 层，叶结点到根结点的路径长度为叶结点的层数）。树的带权路径长度记为WPL = ，n 个权值构成一颗有 n 个叶结点的二叉树，相应的叶结点的路径长度为。可证明哈夫曼树的WPL是最小的。

1. 哈夫曼压缩

哈夫曼压缩是个无损的压缩算法，一般用来压缩文本和程序文件。哈夫曼压缩属于可变代码长度算法一族。意思是个体符号（例如，文本文件中的字符）用一个特定长度的位序列替代。因此，在文件中出现频率高的符号，使用短的位序列，而那些很少出现的符号，则用较长的位序列。

## 1.3相关知识

1. 二进制文件的创建、读取、写入操作
2. 数学统计
3. 哈夫曼树的建立和运用
4. 哈夫曼编码的建立和获取
5. 字节的位运算
6. C++类、文件流相关知识的熟悉掌握。
7. QT界面类创建以及相关知识的熟悉掌握
8. 需求分析

## 2.1 课题调研

随着计算机技术的发展，人们对文件信息的需求量逐渐增大，不光对文件个数的使用量增加，单个文件的内存也增大到几个G ，这种现象的产生，将导致计算机上内存的消耗过大，以及计算机对文件的操作变得十分困难。

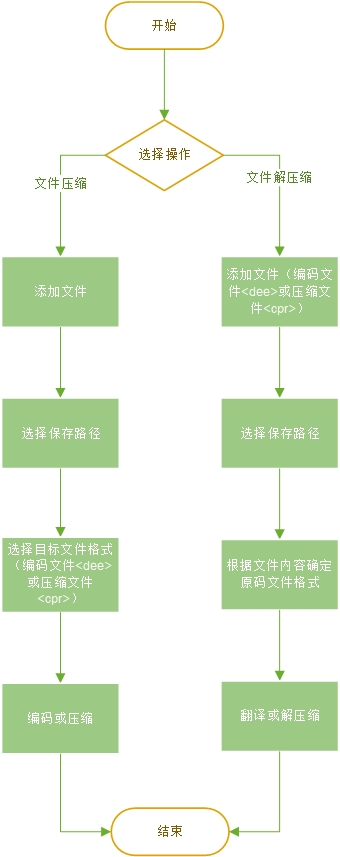
## 2.2 用户需求分析

经过调研，目前用户比较倾向的压缩工具的特点有：使用方便、用户友好，在压缩率以及压缩速度方面表现出色。

在这个背景下，我们小组决定运用哈夫曼算法，设计一款符合大众要求的小压缩工具。

1. 方案设计

## 3.1总体功能设计

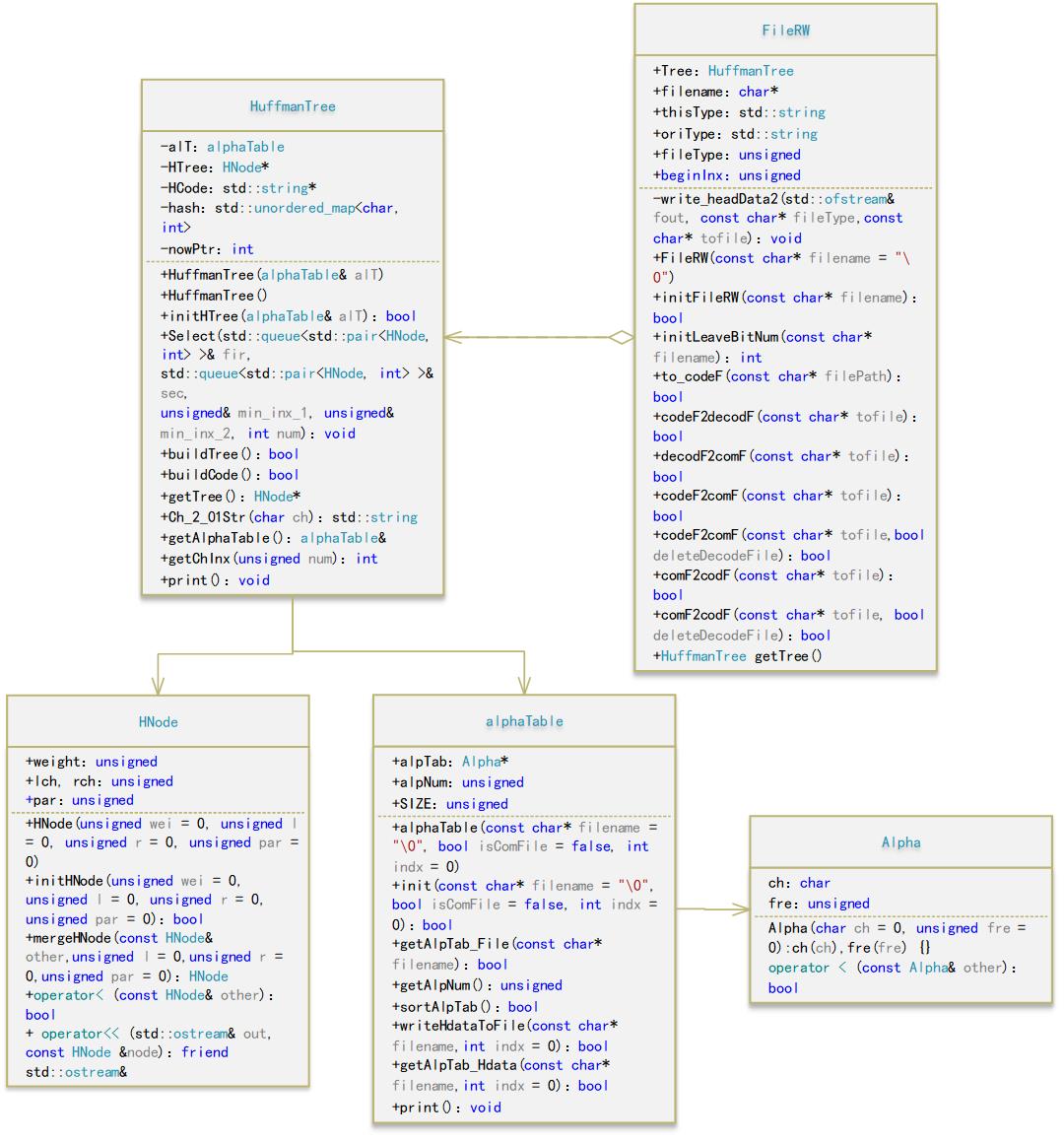


## 3.2数据结构设计

|  |
| --- |
| /\*字母表类：class alphaTable\*/  Alpha\* alpTab; //指向由字母组成的动态数组首地址  unsigned alpNum; //字母数量（种类数）  unsigned SIZE; //字母表大小 |

|  |
| --- |
| /\*哈夫曼树类：class HuffmanTree\*/  alphaTable alT; //建树的字母表  HNode \*HTree; //有哈夫曼结点类组成的静态链表，存储建好的哈夫曼二叉树  std::string \*HCode; //哈夫曼编码表，存储每个字符对应的01编码  std::unordered\_map<char, int>hash; //利用stl建立的字符同在树表索引的映射关系  int nowPtr; //当前指针，解码过程中用到，指向当前正在查找的结点的索引 |

## 3.3类原型设计



|  |
| --- |
| /\*字母结构体组成\*/  typedef struct Alpha{  char ch; //字符  unsigned fre; //当前字符出现频率  Alpha(char ch = 0, unsigned fre = 0):ch(ch),fre(fre) {}  bool operator < (const Alpha& other) {  if (fre != other.fre) return fre < other.fre;  return ch < other.ch;  }  }Alpha; |

|  |
| --- |
| /\*字母表结点类\*/  class alphaTable  {  public:  /\*字母表类\*/  Alpha\* alpTab; //指向由字母组成的动态数组首地址  unsigned alpNum; //字母数量（种类数）  unsigned SIZE; //字母表大小  public:  //字母表构造函数，调用初始化函数初始化  alphaTable(const char\* filename = "\0", bool isComFile = false, int indx = 0);  //字母表初始化函数  bool init(const char\* filename = "\0", bool isComFile = false, int indx = 0);  //从原码文件读取文件建立字母表  bool getAlpTab\_File(const char\* filename);  //获得字母种类数量  unsigned getAlpNum();  //字母表排序函数  bool sortAlpTab();  //向文件中以二进制形式写入头数据（字母种类数量，各字符种类以及出现频率）  bool writeHdataToFile(const char\* filename,int indx = 0);  //从译码文件或压缩文件中读取头数据获得字母表信息  bool getAlpTab\_Hdata(const char\* filename,int indx = 0);  //字母表信息打印函数，用于控制台字母表类测试  void print();  }; |

|  |
| --- |
| /\*哈夫曼树结点类\*/  class HNode  {  public:  unsigned weight; //权重  unsigned lch, rch; //左右孩子结点  unsigned par; //双亲结点  public:  //构造函数，利用初始化函数构造  HNode(unsigned wei = 0, unsigned l = 0, unsigned r = 0, unsigned par = 0);  //初始化函数  bool initHNode(unsigned wei = 0, unsigned l = 0, unsigned r = 0, unsigned par = 0);  //运算符重载  bool operator< (const HNode& other);  //输出流重载，用于控制台后序测试  friend std::ostream& operator<< (std::ostream& out, const HNode &node);  }; |

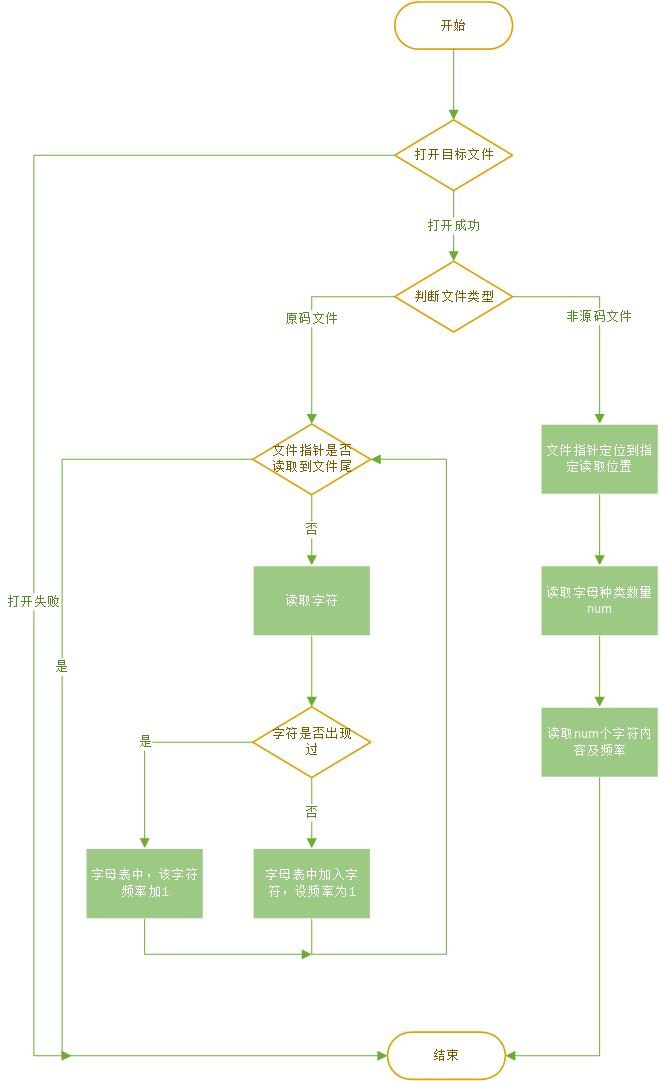
|  |
| --- |
| /\*哈夫曼树类\*/  class HuffmanTree  {  private:  /\*哈夫曼树类\*/  alphaTable alT; //建树的字母表  HNode \*HTree; //有哈夫曼结点类组成的静态链表，存储建好的哈夫曼二叉树  std::string \*HCode; //哈夫曼编码表，存储每个字符对应的01编码  //unsigned hash[270];  std::unordered\_map<char, int>hash; //利用stl建立的字符同在树表索引的映射关系  int nowPtr; //当前指针，解码过程中用到，指向当前正在查找的结点的索引  public:  //构造函数  HuffmanTree(alphaTable& alT);  HuffmanTree();  //初始化函数  bool initHTree(alphaTable& alT);  //挑选结点函数，以及建树函数  void Select(std::queue<std::pair<HNode, int> >& fir, std::queue<std::pair<HNode, int> >& sec,  unsigned& min\_inx\_1, unsigned& min\_inx\_2, int num);  bool buildTree();  //哈夫曼编码表建立函数  bool buildCode();  //获得建立的树表，用于后序控制台测试  HNode\* getTree();  //获得相应字符对应的01编码  std::string Ch\_2\_01Str(char ch);  //获得建树用字母表，用于后序测试，以及文件头数据的写入  alphaTable& getAlphaTable();  //根据输入的01数字，进行当前指向结点（nowPtr）移动。  //获得当前指向结点的索引，若为叶子结点则返回索引，否则返回-1  int getChInx(unsigned num);  //树打印函数，用于后序控制台测试  void print();  }; |

|  |
| --- |
| /\*位操作函数类\*/  class BitDeal  {  public:  //获得对应位的值  unsigned getBit(char& c, int i) ;  //改变对应位的值  void setBit(char& c, int i, int v) ;  }; |

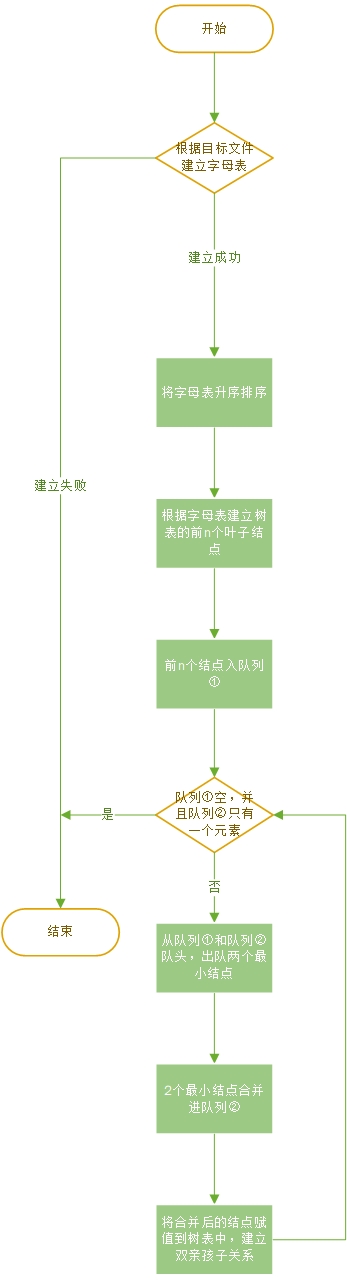
|  |
| --- |
| /\*文件读写类\*/  class FileRW  {  public:  HuffmanTree Tree; //由文件建立的哈夫曼树类  //当前文件类，文件的文件名（路径）  char\* filename;  int leaveBitNum;  //当前文件类型（后缀名）  std::string thisType;  //原码文件类型（后缀名）  std::string oriType;  //当前文件类别，表示是原码文件，还是译码文件，还是压缩后文件  unsigned fileType;  //读写文件时，01编码（压缩码）的起始位置。  unsigned beginInx;  private:  void write\_headData2(std::ofstream& fout, const char\* fileType,const char\* tofile);  public:  FileRW(const char\* filename = "\0");  bool initFileRW(const char\* filename);  int initLeaveBitNum(const char\* filename);  //译码文件和压缩文件，自动转化为原格式的文件  bool to\_codeF(const char\* filePath);  //原码文件和译码文件的转换  bool codeF2decodF(const char\* tofile);  bool decodF2comF(const char\* tofile);  //译码文件和压缩文件的转化  bool comF2decodF(const char\* tofile);  bool decodF2codeF(const char\* tofile);  //原码文件和压缩文件的转化。（可以是经过中间过程的译码文件转换，即第二个重载）。  bool codeF2comF(const char\* tofile);  bool codeF2comF(const char\* tofile,bool deleteDecodeFile);  bool comF2codF(const char\* tofile);  bool comF2codF(const char\* tofile, bool deleteDecodeFile);  //获得建类所使用的的哈弗曼树，用于后序控制台函数测试  HuffmanTree getTree();  }; |

## 3.4主算法设计

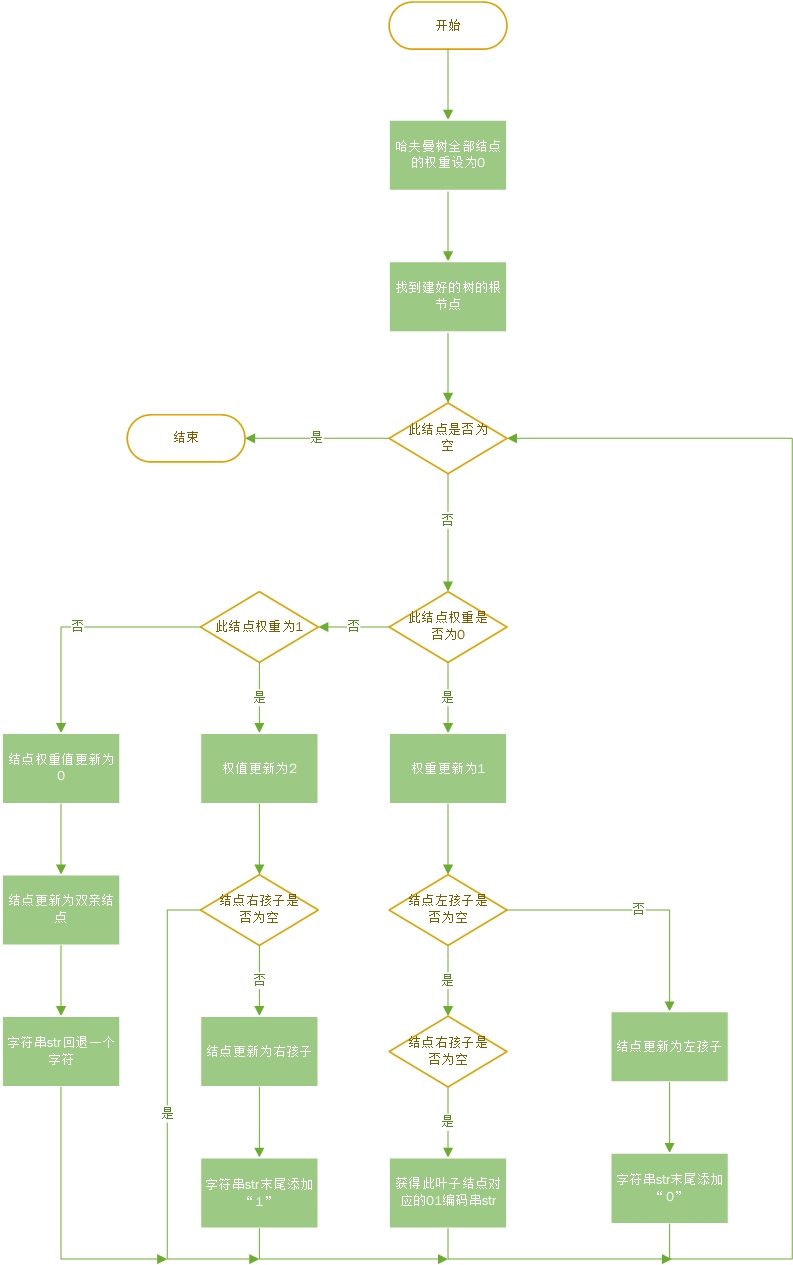
1. 字母表建立算法设计



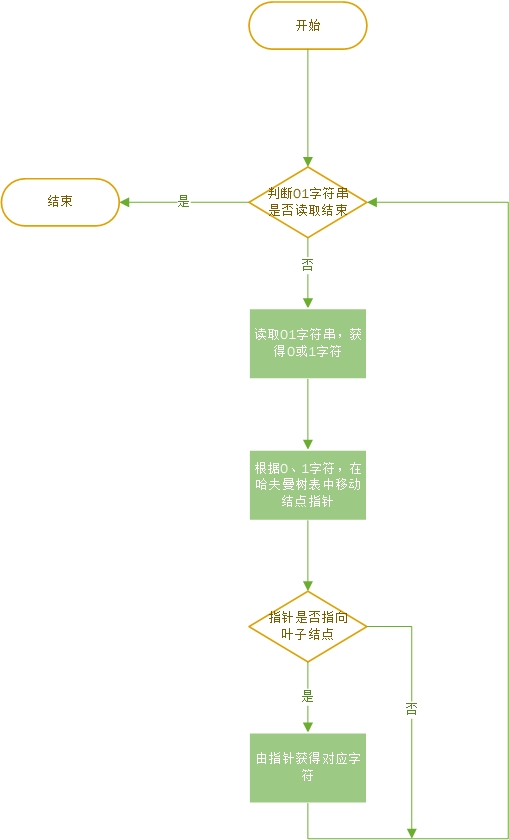
1. 哈弗曼树建立算法设计



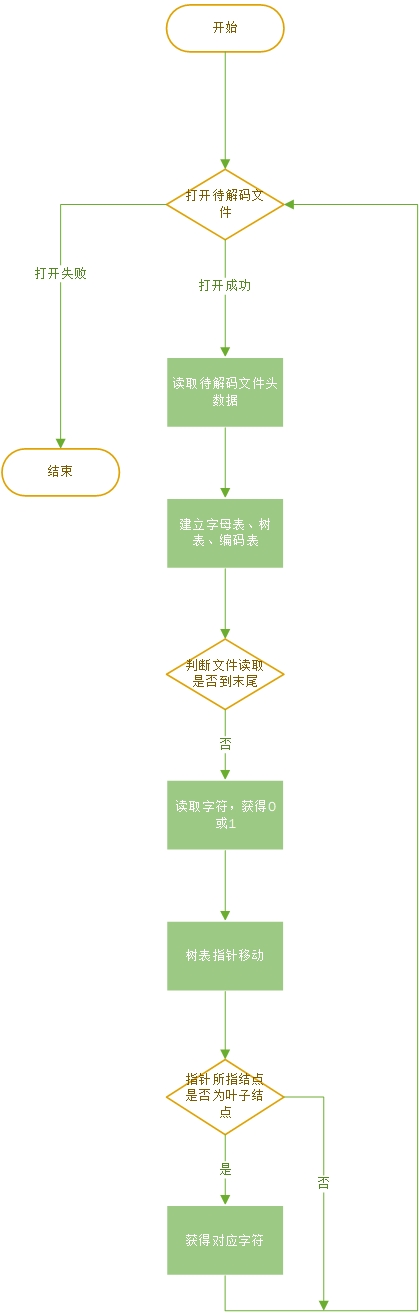
1. 哈夫曼编码表建立算法设计



1. 文件解码算法设计

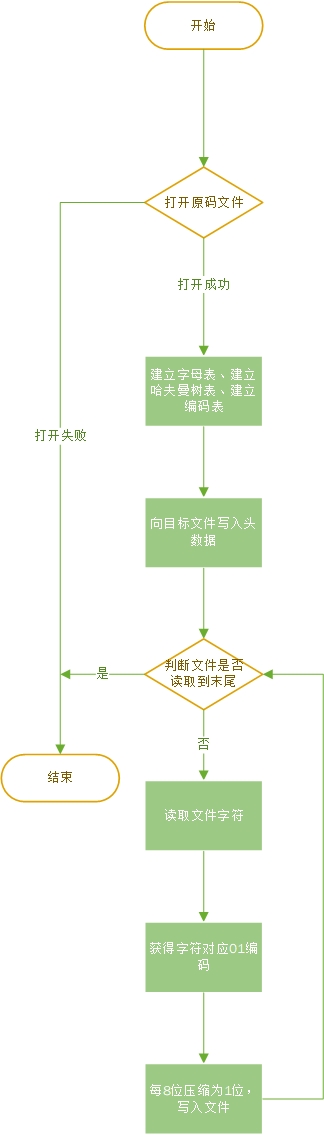


解码算法

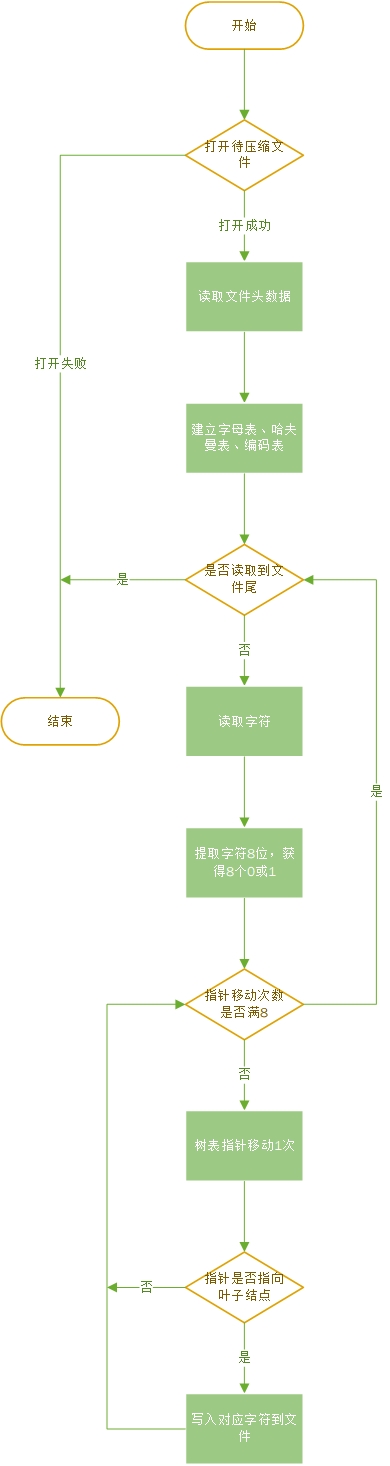


文件解码过程

1. 文件压缩算法设计

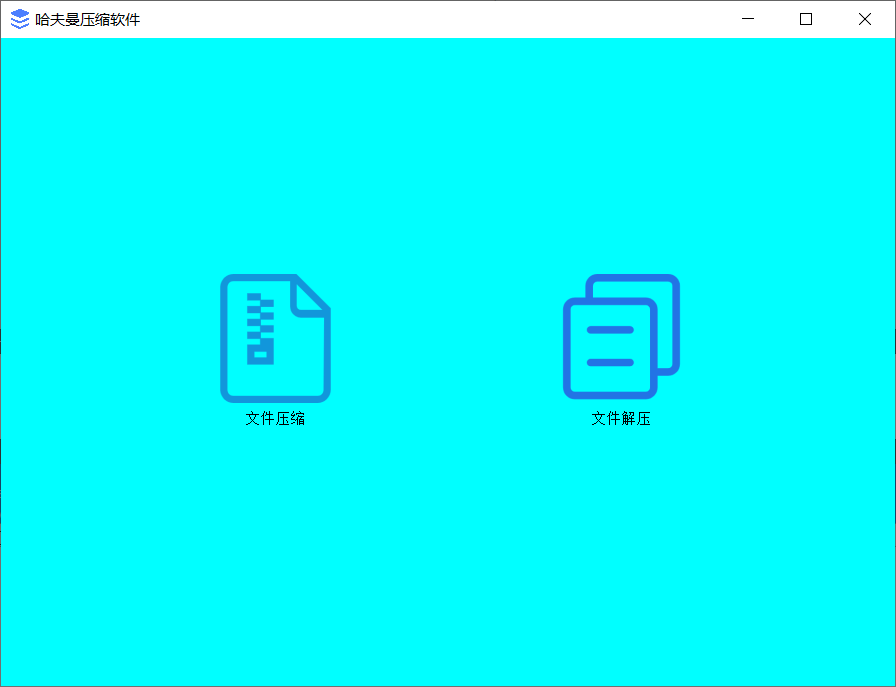


1. 文件解压算法设计

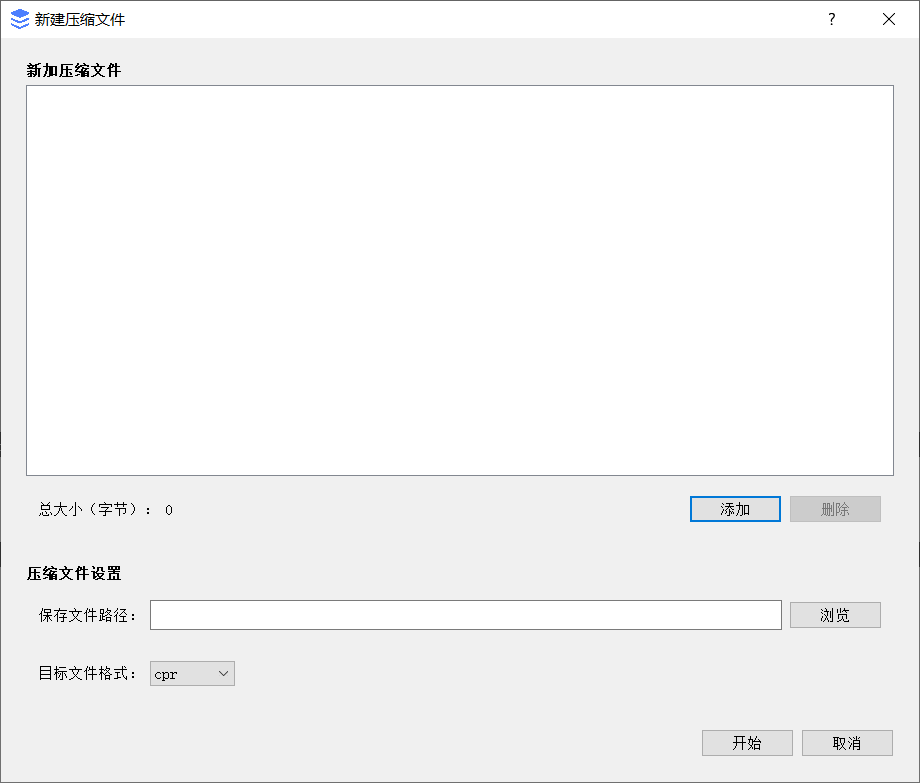


## 3.5用户界面设计

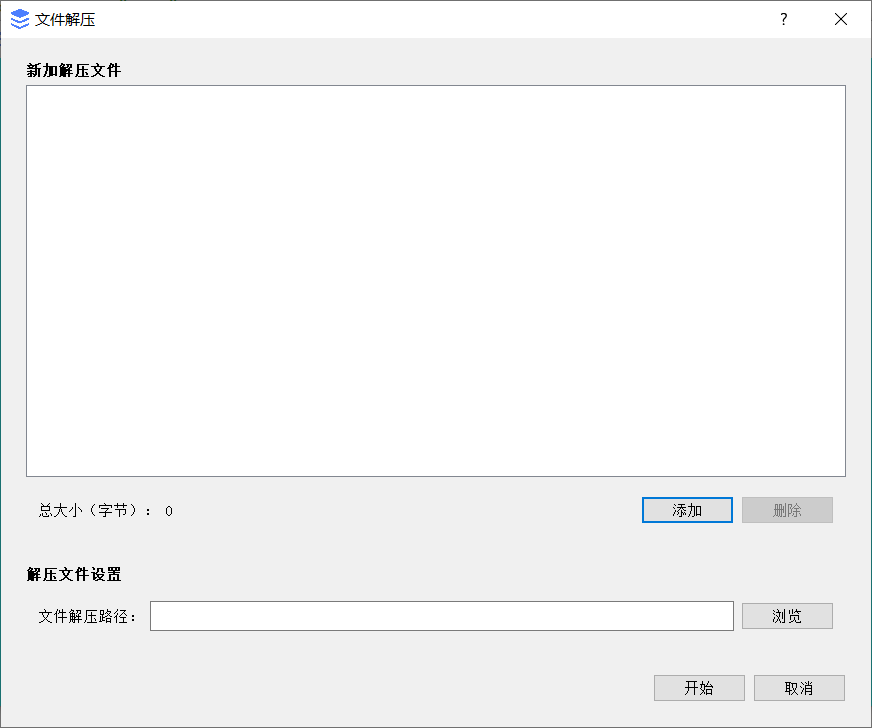
1. 主界面



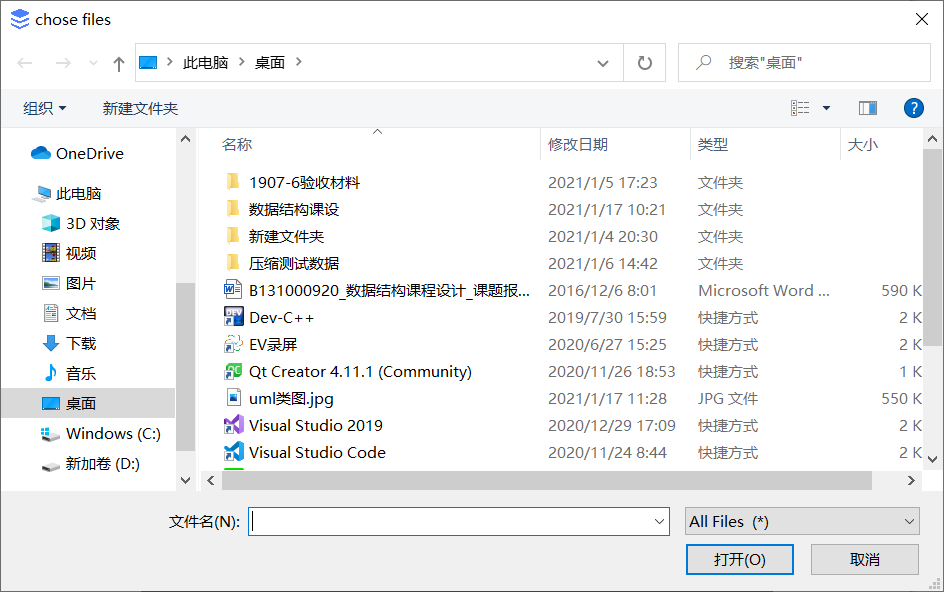
1. 压缩界面

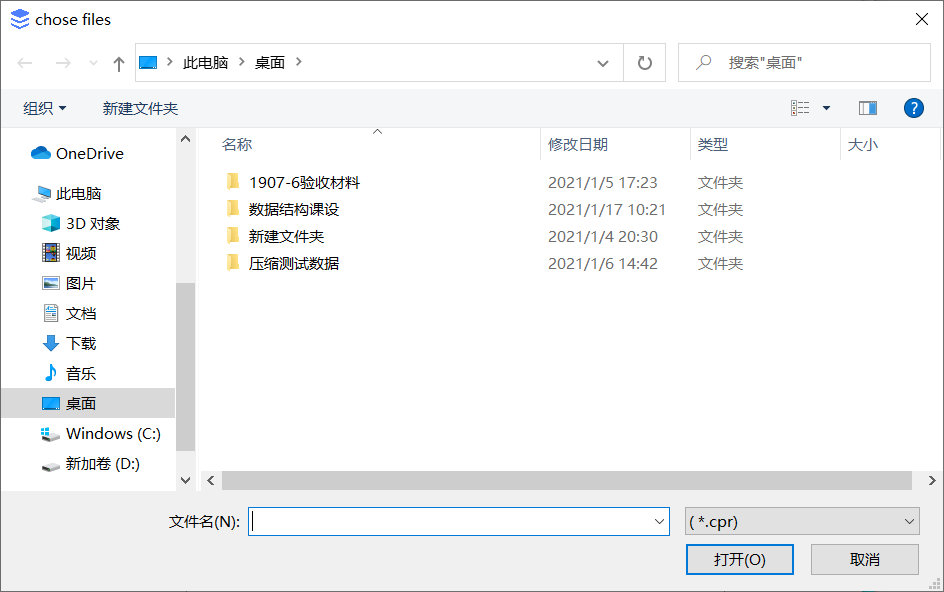


1. 解压界面



1. 查找文件和寻找压缩路径界面





1. 方案实现

## 4.1开发环境与工具

硬件：个人PC

软件：Visual Studio 2019 + Qt Creator 4.11.1 (Community)

开发环境：Windows 10 + VS2019

## 4.2程序设计关键技术

1. 根据原码文件建立字母表
2. 根据字母表在O(nlogn)时间复杂度内建立哈夫曼树
3. 根据哈夫曼树建立和字母表对应的哈夫曼编码表
4. 根据连续的01的编码获得原字符串
5. 译码文件和压缩文件中头数据的写入和读取
6. 实时读取实时编（解）码实时文件写入
7. 实时读取实时压缩（解压缩）实时文件复原
8. QT界面类库函数的调用

## 4.3个人设计实现

### 4.3.1白千一个人设计实现

1. 建立字母表

|  |
| --- |
| //读取原码文件获得字母表  bool alphaTable::getAlpTab\_File(const char\* filename)  {  //利用STL建立字符同字符出现频率的映射  std::unordered\_map<char, int>hash;  //打开原码文件  std::ifstream fin(filename, std::ios::in | std::ios::binary);  if (!fin.is\_open()) return false;  //读取字符，建立字母表  char ch;  while (!fin.eof()) {  fin.read(&ch, sizeof(ch));  if (hash[ch] == 0) {  alpNum++;  //若字母种类超出字母表大小，则扩展  if (alpNum >= SIZE) {  Alpha\* temp = new Alpha[SIZE + Size];  for (unsigned i = 1; i < SIZE; ++i) {  temp[i].ch = alpTab[i].ch;  temp[i].fre = alpTab[i].fre;  }  Alpha\* m = alpTab; alpTab = temp; delete m;  SIZE += Size;  }  if (alpNum < SIZE) {  alpTab[alpNum].fre = 1;  alpTab[alpNum].ch = (ch);  }  hash[ch] = alpNum;  //alpNum++;  }  else {  alpTab[hash[ch]].fre++;  }  }  return true;  } |

|  |
| --- |
| //读取头数据获得字母表  bool alphaTable::getAlpTab\_Hdata(const char\* filename, int indx)  {  //打开目标文件  std::ifstream fin(filename, std::ios::in | std::ios::binary);  if(!fin.is\_open()) return false;  //定位到读取头数据位置  fin.seekg(indx);  fin.read((char\*)&alpNum, sizeof(alpNum));  //扩展大小  if (alpNum > SIZE-1) {  alpTab = new Alpha[alpNum + Size];  }  //读取字符种类和对应频率  for (unsigned i = 1; i <= alpNum; ++i) {  fin.read((char\*)&alpTab[i].ch, sizeof(char));  fin.read((char\*)&alpTab[i].fre, sizeof(unsigned));  }  fin.close();  return true;  } |

1. 建立哈夫曼编码表

|  |
| --- |
| bool HuffmanTree::buildCode()  {  //只有一个字母，特殊情况特殊考虑  if (alT.alpNum == 1) {  HCode[1] = "0";  return true;  }  //非递归遍历二叉树，获得每个叶子结点对应的01编码  std::string str;  unsigned root = 2 \* (int)alT.alpNum - 1;  for (unsigned i = 1; i <= root; ++i) {  HTree[i].weight = 0;  }  unsigned p = root;  while (p) {  if (HTree[p].weight == 0) {  HTree[p].weight = 1;  if (HTree[p].lch != 0) {  p = HTree[p].lch;  str.push\_back('0');  }  else if (HTree[p].rch == 0) {  HCode[p] = str;  }  }  else if (HTree[p].weight == 1) {  HTree[p].weight = 2;  if (HTree[p].rch != 0) {  p = HTree[p].rch;  str.push\_back('1');  }  }  else {  HTree[p].weight = 0;  p = HTree[p].par;  if (!str.empty()) str.pop\_back();  }  }  return true;  } |

1. 文件压缩

|  |
| --- |
| bool FileRW::codeF2comF(const char\* tofile)  {  if (fileType != 1) return false;  std::ofstream fout(tofile, std::ios::out | std::ios::binary); //打开目标文件  if (!fout.is\_open()) return false;  std::ifstream fin(filename, std::ios::in | std::ios::binary); //打开原码文件  if (!fin.is\_open()) return false;  write\_headData2(fout, "cpr", tofile); //写入头数据  fout.seekp(beginInx); //定位到压缩码开始写入的位置  //开始译码并压缩  BitDeal BD;  char ch;  char value;  int bitnum = 1;  while (fin.read(&ch,sizeof(char))) {  std::string code = this->Tree.Ch\_2\_01Str(ch); //获得字符对应的01编码  for (const auto& it : code) {  BD.setBit(value, bitnum, it - '0'); //修改第bitnum位的值  if (bitnum >= 8) {//修改满8位，写入到目标文件  fout.write(&value, sizeof(value));  bitnum = 1;  }  else bitnum++;  }  if (fin.peek() == EOF) {//读取到文件尾  //计算冗余位数  if (bitnum == 1) leaveBitNum = 0;  else {//如果即将改变的位序超过1，将最后几个价值位写入，并计算冗余位  fout.write(&value, sizeof(value));  leaveBitNum = 9 - bitnum;  }  break;  }  }  fin.close();  fout.close();  return true;  } |

1. 文件解压

|  |
| --- |
| bool FileRW::comF2codF(const char\* tofile)  {  if (fileType != 2) return false;  //打开文件  std::ofstream fout(tofile, std::ios::out | std::ios::binary);  if (!fout.is\_open()) return false;  std::ifstream fin(filename, std::ios::in | std::ios::binary);  if (!fin.is\_open()) return false;  fin.seekg(beginInx); //定位到原码写入位置  BitDeal BD;  char ch;  //bitnum表示要读取的第几位  int bitnum = 9;  int num;  //未到文件尾，或者最后一个字节仍有价值位未读取  while ( fin.peek() != EOF || (bitnum <= (int)sizeof(char)\*8 - leaveBitNum)) {  if (bitnum > 8) {//说明一个char类型已经读完，该读入新的字符了  fin.read(&ch, sizeof(char));  bitnum = 1;  }  num = BD.getBit(ch, bitnum++);  int inx = Tree.getChInx(num);  if (inx != -1) {  fout.write(&Tree.getAlphaTable().alpTab[inx].ch, sizeof(char));  }  }  fin.close();  fout.close();  return true;  } |

1. 压缩界面的编写

|  |
| --- |
| class **ComDialog** : public QDialog  {  Q\_OBJECT  public:  explicit **ComDialog**(QWidget \*parent = nullptr);  ~***ComDialog***();  private slots:  void **on\_addFileNamesButton\_clicked**(); //点击“添加”按钮  void **on\_delteFileNamesButton\_clicked**(); //点击“删除”按钮  void **on\_choseFilePathButton\_clicked**(); //点击“浏览”按钮  void **on\_CanelButton\_clicked**(); //点击“取消”按钮  void **on\_StartButton\_clicked**(); //点击“开始”按钮  private:  Ui::ComDialog \*ui;  }; |

### 4.3.2金忠信个人设计实现

1. 建立哈弗曼树

|  |
| --- |
| bool HuffmanTree::buildTree()  {  //设置两个初始队列  std::queue<std::pair<HNode, int> > fir;  std::queue<std::pair<HNode, int> > sec;  //将叶子结点按次序入第一个队列  for (unsigned i = 1; i <= alT.alpNum; ++i) {  fir.push(std::make\_pair(HTree[i], i));  }  unsigned min1, min2; //最小的两个元素在树表中对应的索引  if (alT.alpNum < 1) { std::cout << "\n字母数量少" << std::endl; return false; };  if (alT.alpNum == 1) return true;  int num = alT.alpNum + 1;  while (!(fir.empty() && sec.size() == 1)) {  Select(fir, sec, min1, min2, num); //挑选出两个最小的结点（索引）  HTree[num] = sec.back().first;  HTree[min1].par = num; HTree[min2].par = num; //建立孩子关系  num++;  }  return true;  } |

1. 树表结点挑选函数

|  |
| --- |
| void HuffmanTree::Select(std::queue<std::pair<HNode, int> >& fir, std::queue<std::pair<HNode, int> >& sec,  unsigned& min\_inx\_1, unsigned& min\_inx\_2, int num)  {  if (fir.empty() && sec.empty()) exit(-1);  //初始化  int min\_weight\_1 = INF, min\_weight\_2 = INF;  min\_inx\_1 = min\_inx\_2 = 0;  //获得第一个队列的队头结点  if (!fir.empty()) {  min\_weight\_1 = fir.front().first.weight;  min\_inx\_1 = fir.front().second;  }  //获得第二个队列的队头结点  if (!sec.empty()) {  min\_weight\_2 = sec.front().first.weight;  min\_inx\_2 = sec.front().second;  }  //如果第一个队列队头结点的权重比第二个小  if (min\_weight\_1 <= min\_weight\_2) {  //第一个结点肯定是第一个队列队头结点了，出队列  fir.pop();  //第二小结点可能是第二个队列队头结点，有可能是第一个队列第二个结点，也就是此刻的一队列队头结点  int weight = INF;  if (!fir.empty()) weight = fir.front().first.weight;  if (weight <= min\_weight\_2) {  //第二小的是一队列第二个结点，出队列  min\_weight\_2 = weight;  min\_inx\_2 = fir.front().second;  fir.pop();  }  else {  //第二小是二对列队头结点，出队列  sec.pop();  }  }  else {//如果第二个队列队头结点的权重比第一个小，道理同上  sec.pop();  int weight = INF;  if (!sec.empty()) weight = sec.front().first.weight;  if (weight <= min\_weight\_1) {  //默认min\_inx\_1为最小的那个，进行交换  min\_weight\_1 = min\_weight\_2; min\_weight\_2 = weight;  min\_inx\_1 = min\_inx\_2; min\_inx\_2 = sec.front().second;  sec.pop();  }  else{  //默认min\_inx\_1为最小的那个，所以利用STL进行交换  std::swap(min\_weight\_1, min\_weight\_2);  std::swap(min\_inx\_1, min\_inx\_2);  fir.pop();  }  }  //将最小的两个结点合并后，入二队列  sec.push(std::make\_pair(HNode(min\_weight\_1 + min\_weight\_2, min\_inx\_1, min\_inx\_2),  num));  } |

1. 主界面的编写

|  |
| --- |
| class **MainWindow** : public QMainWindow  {  Q\_OBJECT  public:  **MainWindow**(QWidget \*parent = nullptr);  ~***MainWindow***();  private slots:  void **on\_deButton\_clicked**(); //点击“文件压缩”按钮  void **on\_pushButton\_2\_clicked**(); //点击“文件解压”  private:  Ui::MainWindow \*ui;  }; |

1. 解压界面的编写

|  |
| --- |
| class **DecomDialog** : public QDialog  {  Q\_OBJECT  public:  explicit **DecomDialog**(QWidget \*parent = nullptr);  ~***DecomDialog***();  private slots:  void **on\_addFileNamesButton\_2\_clicked**(); //点击“添加”按钮  void **on\_delteFileNamesButton\_2\_clicked**(); //点击“删除”按钮  void **on\_choseFilePathButton\_2\_clicked**(); //点击“浏览”按钮  void **on\_CanelButton\_2\_clicked**(); //点击“取消”按钮  void **on\_StartButton\_2\_clicked**(); //点击“开始”按钮  private:  Ui::DecomDialog \*ui;  }; |

1. 测试和调试

## 5.1个人测试

### 5.1.1白千一测试

1. 位运算函数类测试

|  |
| --- |
| 1. 编写测试程序   //位运算函数类测试  void BitDeal\_test()  {  BitDeal BD;  char a = 3;  cout << "初始位序列为：";  for (int i = 8; i >= 1; --i)  std::cout << BD.getBit(a, i);  std::cout << "\n请输入要改变的位和对应的改变值:\n";  int bitnum = 1, v = 0;  while (cin >> bitnum >> v) {  if (bitnum < 0 || bitnum > 8) break;  if (v != 0 && v != 1) break;  BD.setBit(a, bitnum, v);  cout << "改变后位序列为：";  for (int i = 8; i >= 1; --i)  std::cout << BD.getBit(a, i);  cout << "\n此字节代表的值为：" << a << endl;  std::cout << "\n请输入要改变的位和对应的改变值:\n";  }  }   1. 运行程序观察运行结果     从运行结果来看，每次都正确的改变了每个位值。位运算函数类的编写是正确的。 |

1. 哈夫曼树表以及编码表建立函数测试

|  |
| --- |
| 1. 编写哈夫曼测试程序   //哈夫曼树建立函数测试  void HuffmanTree\_test()  {  HuffmanTree tree(\*new alphaTable("1234.txt")); //根据目标文件建立字母表，建立树表，建立编码表  tree.print(); //将树打印观察结果  //输入字符，测试编码表建立是否成功，以及测试字符同索引的映射是否建立成功  char ch;  std::cout << "请输入字符：";  while (cin >> ch) {  if (ch == '#') break;  cout << tree.Ch\_2\_01Str(ch) << endl;  std::cout << "请输入字符：";  }  }   1. 运行测试函数比较结果       从程序运行结果来看，哈夫曼树表和编码表都能建立成功，并且建立起了字符同01编码串之间的映射关系。 |

1. 文件压缩解压测试

|  |
| --- |
| 1. 编写文件压缩解压测试程序   void FileRW\_codeF\_comF\_test()  {  FileRW file("1234.txt"); //根据原码文件建立字母表，哈夫曼树表以及编码表  file.codeF2comF("1234.cpr"); //调用压缩函数  cout << "根据原码文件建立的相关数据：" << endl;  cout << "冗余位数：" << file.leaveBitNum << endl;  cout << "此文件类型：" << file.thisType << endl;  cout << "原码文件类型：" << file.oriType << endl;  file.getTree().getAlphaTable().print();  FileRW file2("1234.cpr"); //根据压缩文件建立  cout << "根据压缩文件建立的相关数据：" << endl;  cout << "冗余位数：" << file2.leaveBitNum << endl;  cout << "此文件类型：" << file2.thisType << endl;  cout << "原码文件类型：" << file2.oriType << endl;  file2.getTree().getAlphaTable().print();  file2.comF2codF("1234\_fromcpr.txt"); //调用解压函数  }   1. 运行程序观察运行结果      1. 观察实际读写文件结果     原码文件以及解压缩后文件      压缩文件以及其内容    解压缩后文件内容  从控制台返回的结果以及实际文件读写的结果来看，原码文件和解压缩文件内容是相同的，文件的压缩解压过程都是正确的。 |

### 5.1.2金忠信测试

1. 字母表建立函数测试

|  |
| --- |
| 1. 编写字母表测试程序   //“建立字母表”测试函数  void alphaTable\_test()  {  alphaTable al("1234.txt"); //读取文本文件，建立字母表  al.print(); //打印字母表，观察出现字符  al.writeHdataToFile("3456.txt"); //检查“头数据写入”函数  alphaTable a2("3456.txt", true); //读取非原码文件，建立字母表  a2.print(); //打印字母表，同原文本文件比对  }   1. 设置原码文本文件        1. 运行测试函数比较结果     可以看出根据原码文件和头数据文件建立字母表是相同的，说明字母表建立的相关函数是正确的。 |

1. 文件编码译码测试

|  |
| --- |
| 1. 编写文件编码译码测试程序   //文件编码译码函数测试  void FileRW\_codeF\_decodF\_test()  {  FileRW file2("1234.txt"); //根据原码文件建立字母表，哈夫曼树表以及编码表  file2.codeF2decodF("1234.dee"); //调用编码函数  cout << "根据原码文件建立的相关数据：" << endl;  cout << "冗余位数：" << file2.leaveBitNum << endl;  cout << "此文件类型：" << file2.thisType << endl;  cout << "原码文件类型：" << file2.oriType << endl;  file2.getTree().getAlphaTable().print();  FileRW file("1234.dee"); //根据译码文件，建立相关数据  if (file.decodF2codeF("1234\_fromdee.txt")) {//译码成功  cout << "根据译码文件建立的相关数据：" << endl;  cout << "冗余位数：" << file.leaveBitNum << endl;  cout << "此文件类型：" << file.thisType << endl;  cout << "原码文件类型：" << file.oriType << endl;  file.getTree().getAlphaTable().print();  }  }   1. 运行测试程序观察运行结果      1. 观察读写文件结果     原码文件以及由译码复原的文件      生成的编码文件    原码文件内容    编码复原文件内容  从控制台返回的结果以及实际文件读写的结果来看，原码文件和编码复原文件内容是相同的，文件的编码解码过程都是正确的。 |

## 5.2组装与系统测试

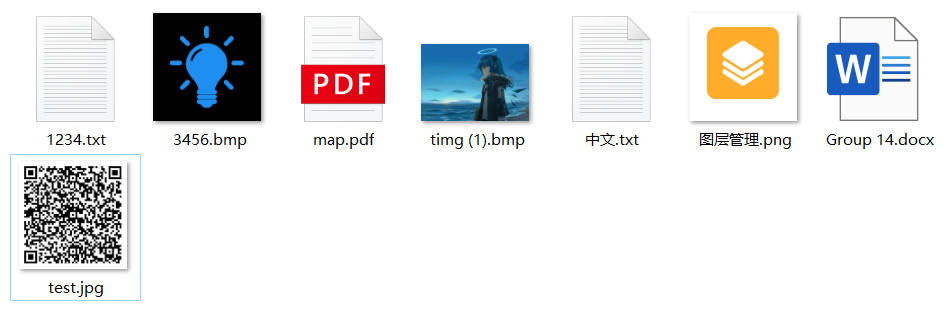
1. 组装

在已经创建好的vs项目文件目录里，创建QT项目文件夹，将编写好的.h和.cpp文件加入到qt项目中，并继续加入编写好的qt界面文件，完成系统的组装。

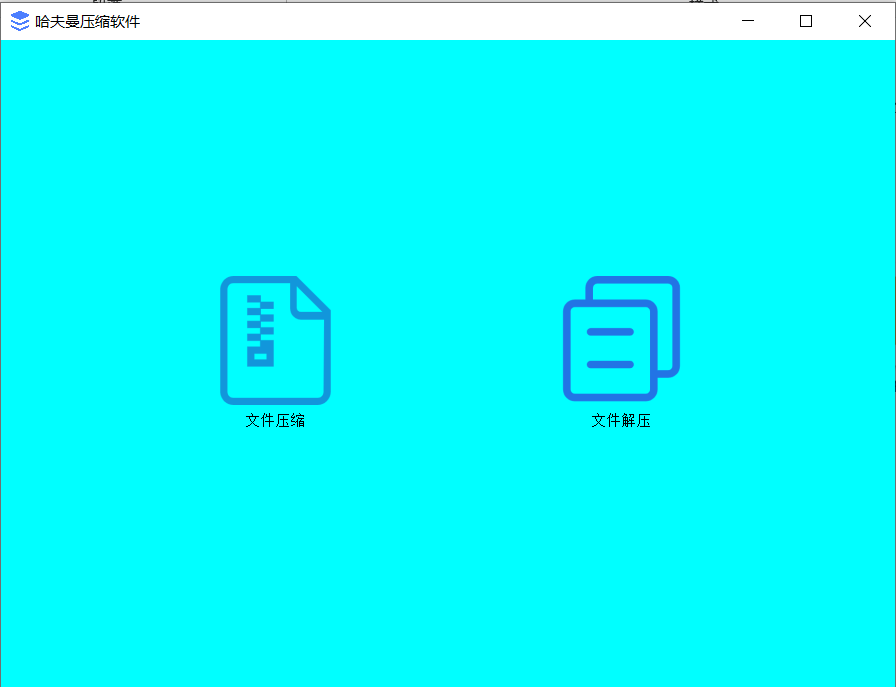
1. 系统测试

文件压缩测试：

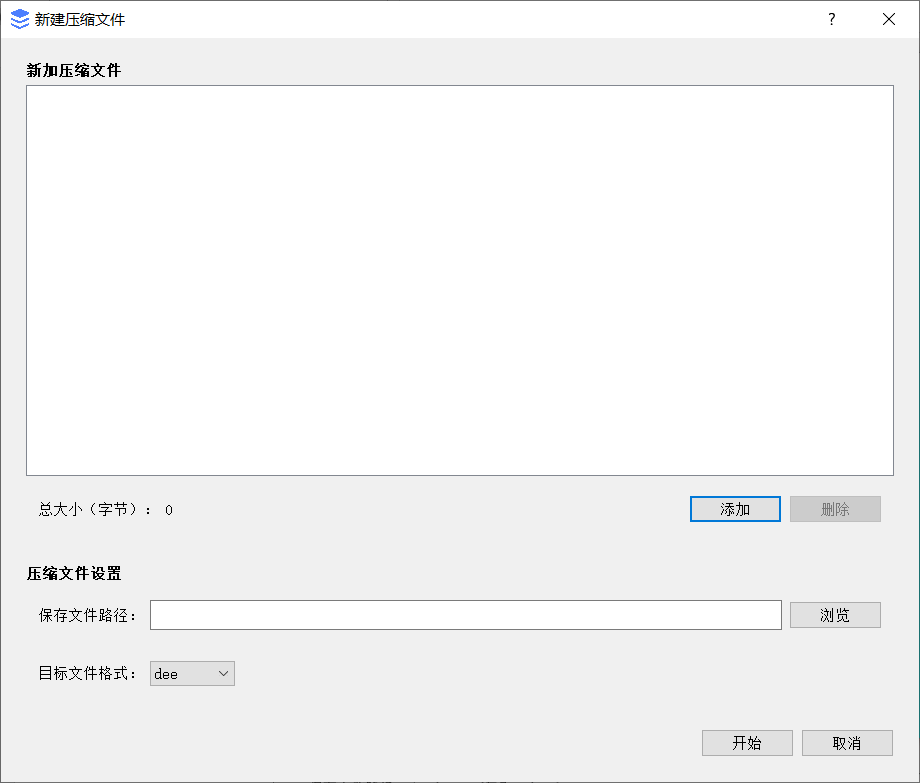
1. 准备不同格式的测试文件



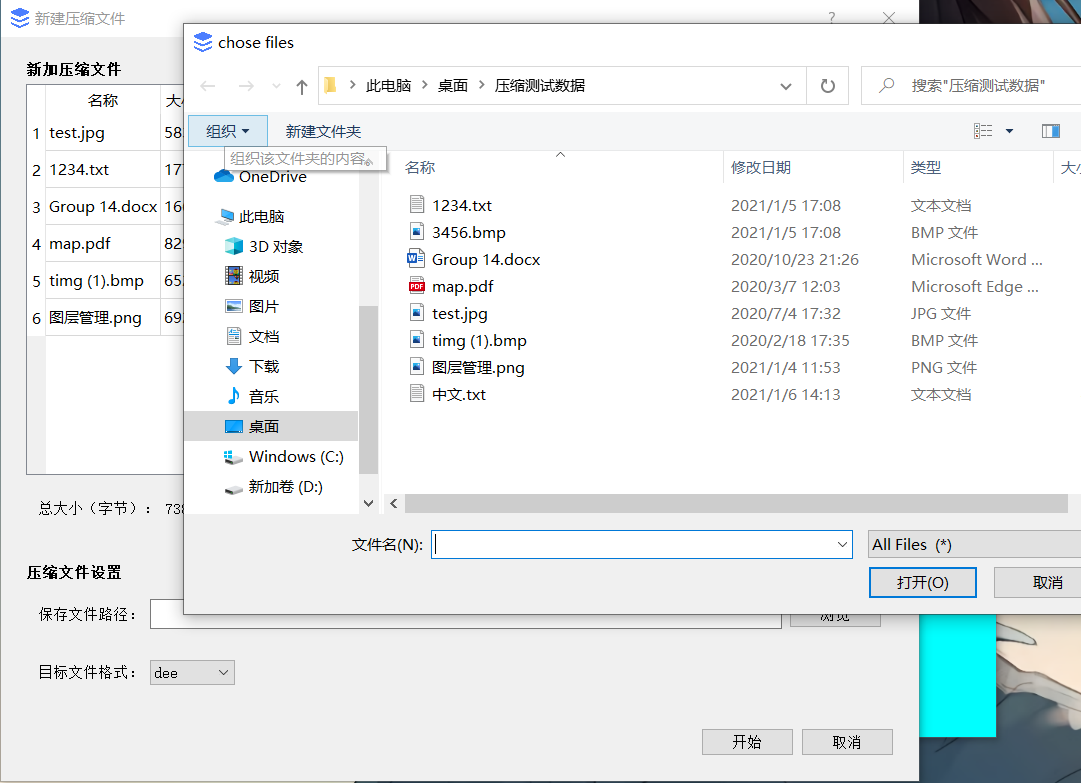
1. 进入主界面



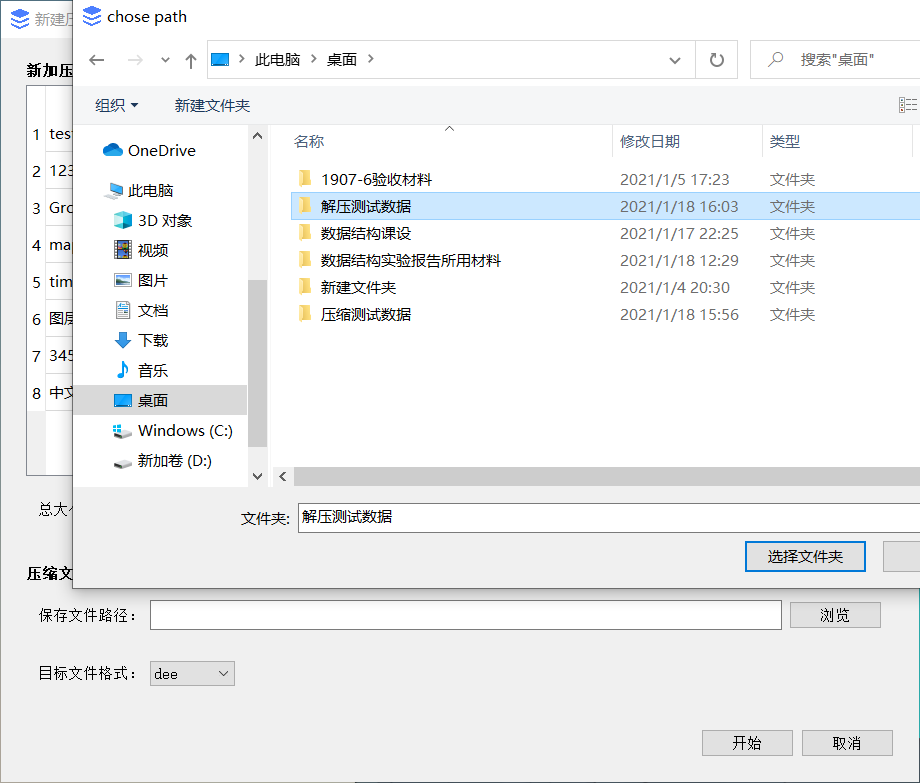
1. 进入文件压缩界面

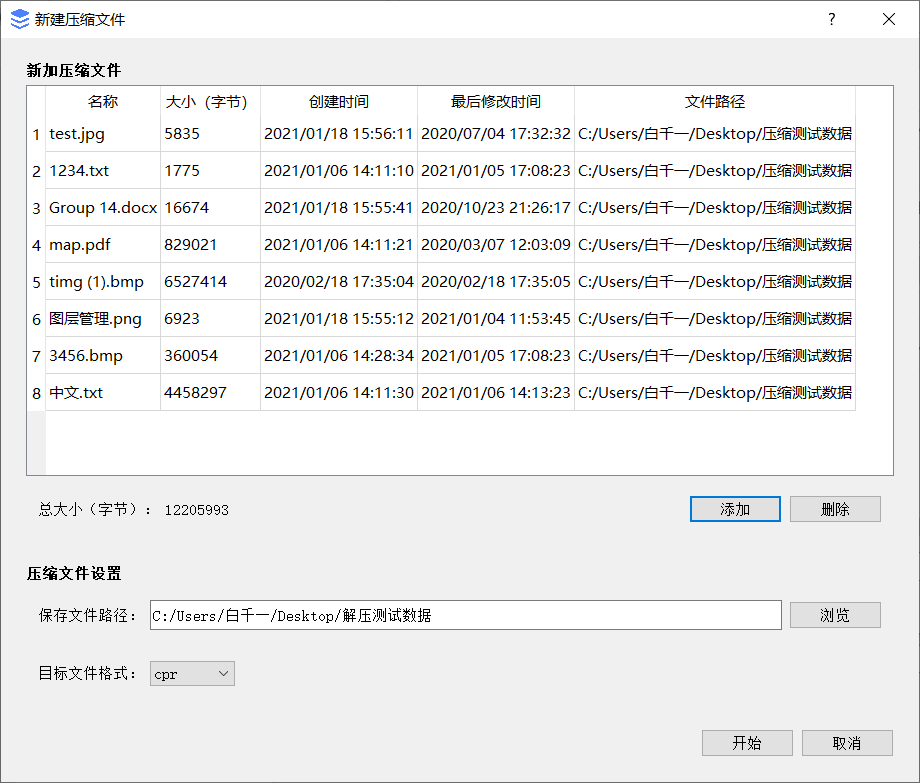


1. 批量添加压缩文件

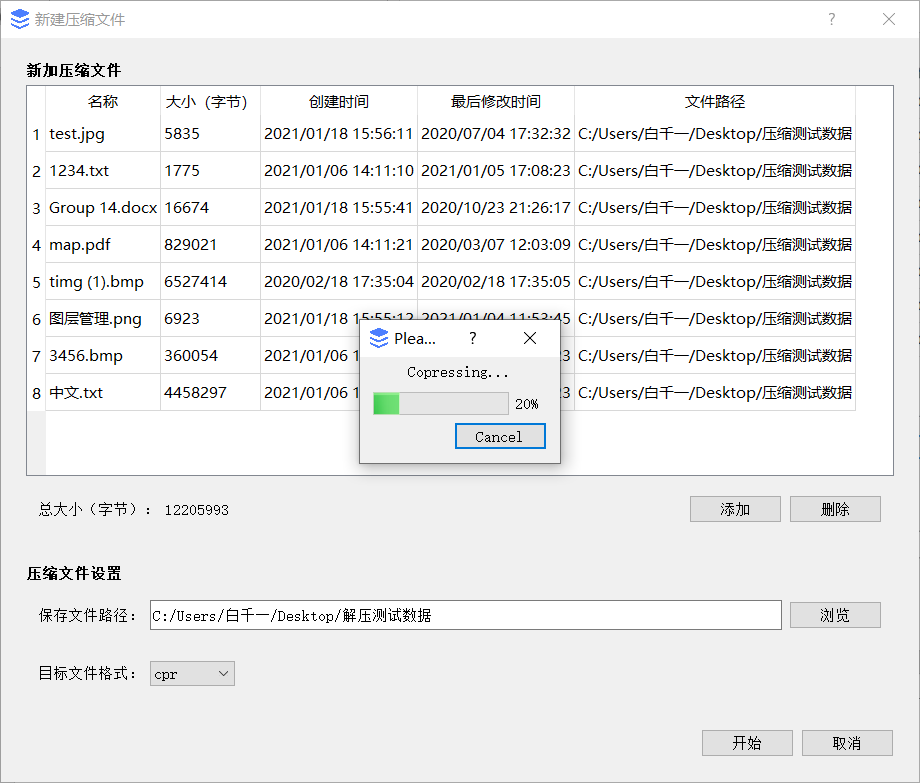


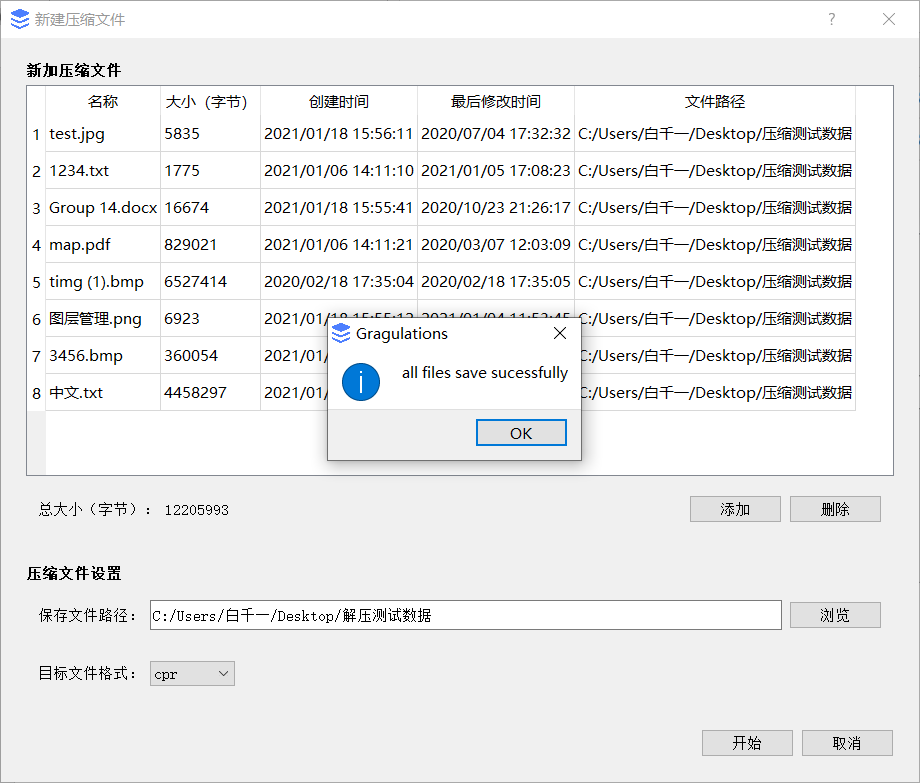
1. 选择保存文件路径以及目标文件格式



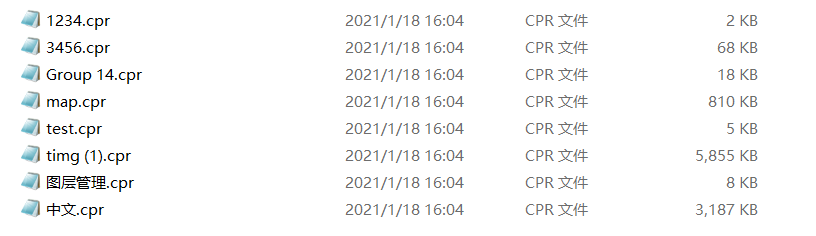


1. 进行压缩



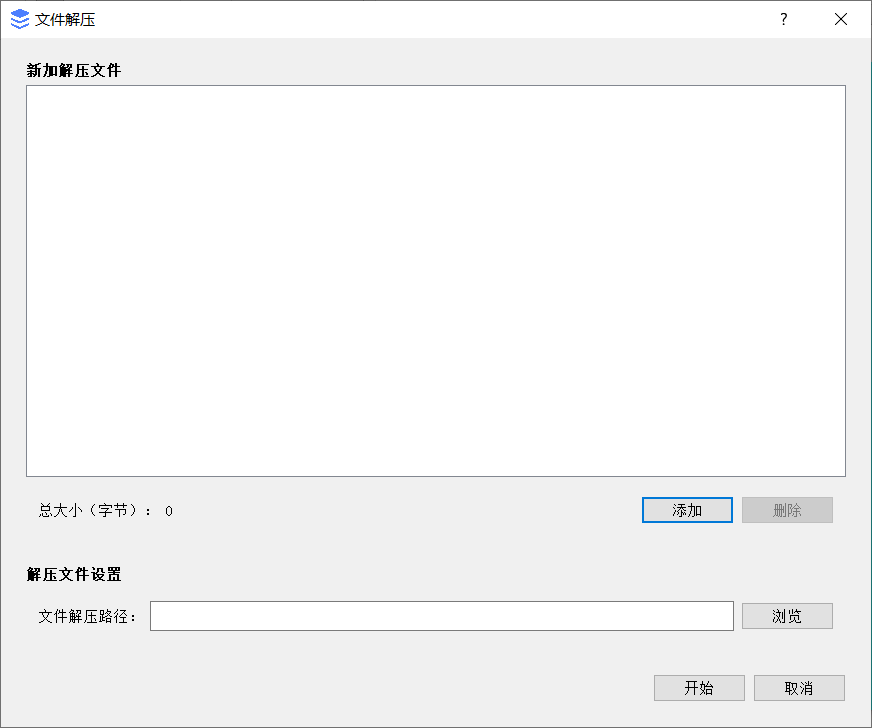


1. 观察文件压缩结果

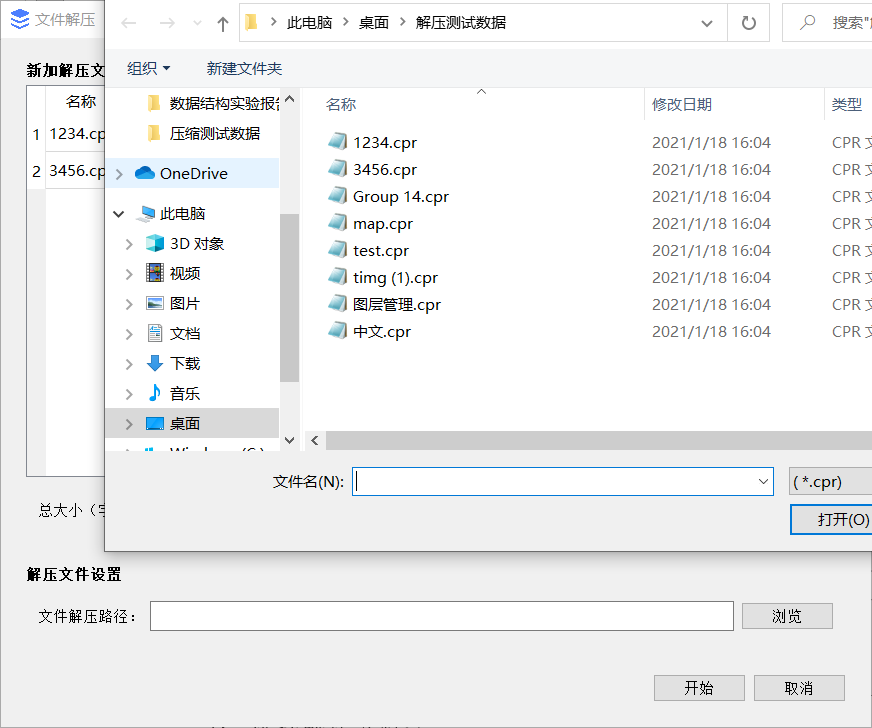


文件解压测试

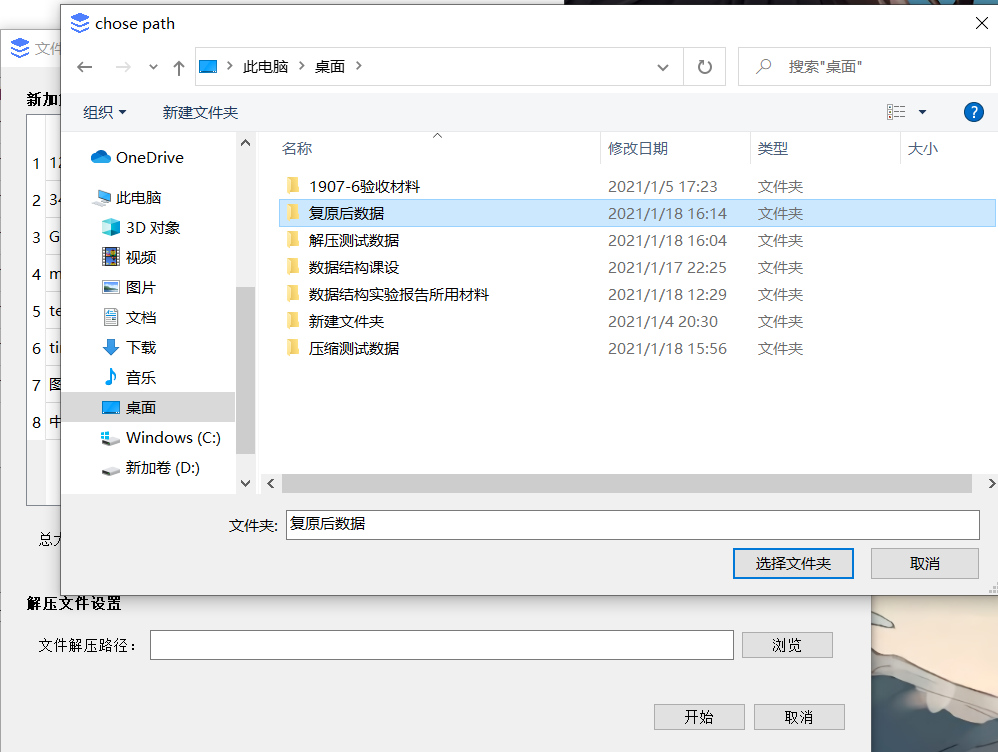
1. 进入解压界面

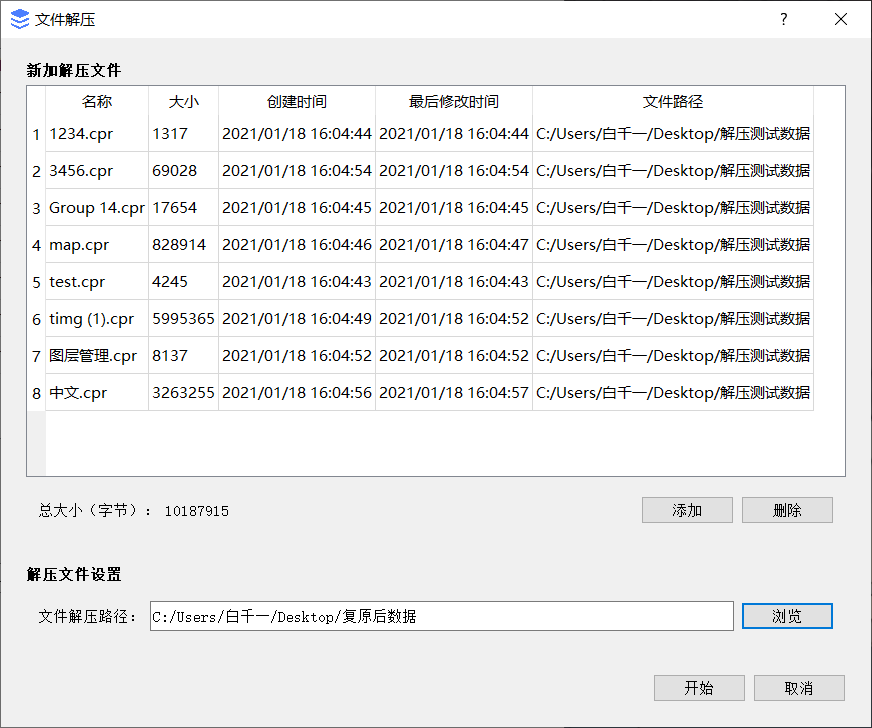


1. 选择解压文件

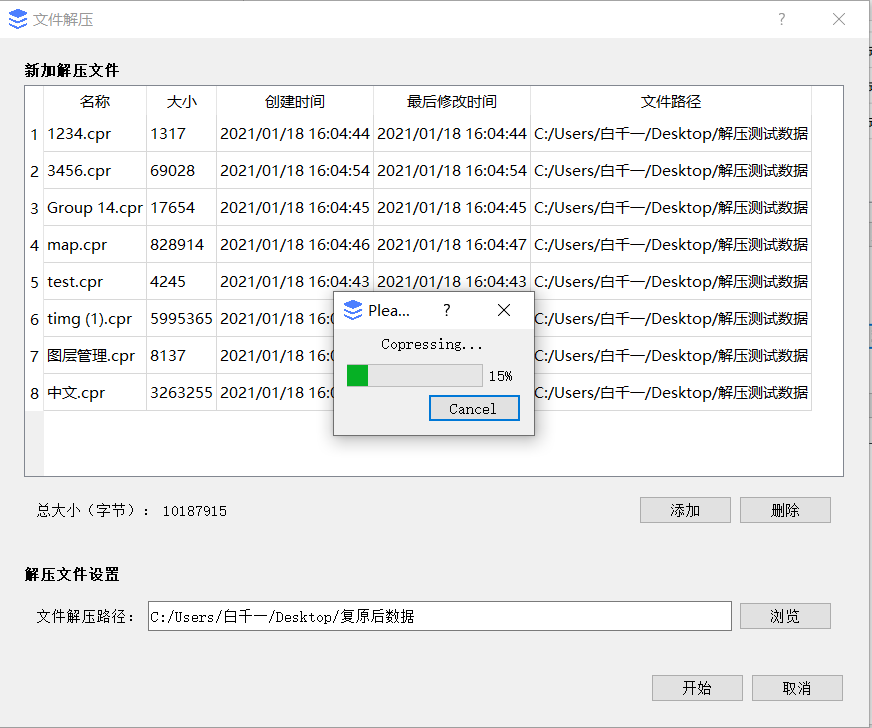


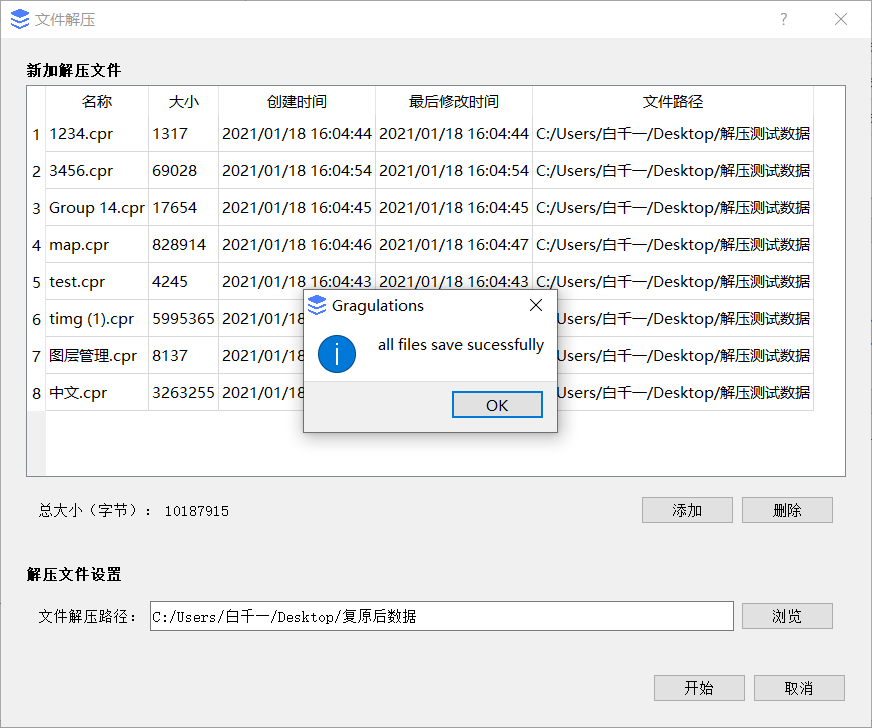
1. 选择解压文件路径



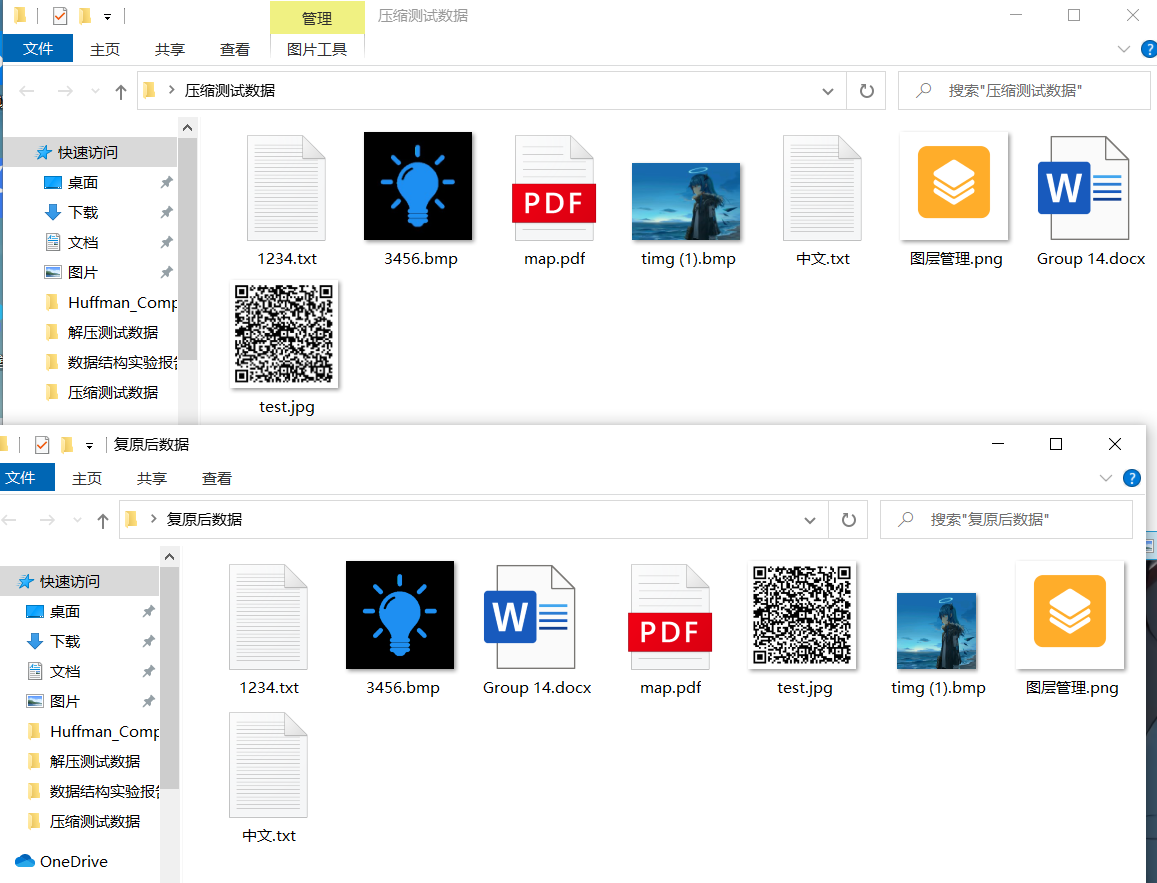


1. 开始解压





1. 比较复原后数据同原数据



可以看出原文件同解压后文件相同，系统测试成功。

1. 课题总结

## 6.1课题评价

本次课题要求采用哈夫曼最优二叉树的相关知识，实现哈夫曼数据压缩技术，设计一个哈夫曼压缩软件。小组设计压缩软件小巧，使用方便、界面工整友好。经过多次验证，小组设计的软件在压缩率方面表现出色，且采取无损压缩，不会产生信息丢失，但压缩解压的速度较慢，有待进一步提高。另外，相比市面上使用率较高的压缩软件，课题小组设计的软件存在无法文件预览、无法创建压缩文件夹等功能过于单一的问题。除此之外，软件仅围绕一个哈夫曼算法展开，没有结合其他优秀的压缩算法，导致软件的实用价值并不高。

## 6.2团队协作

整个工程分工细致，任务明确，在组长的带领下，小组全体组员出色的完成了自己的任务。经过各个成员的反复修改，小组程序的错误越来越少，并在验收中一次通过。在任务完成过程中，团队成员的团队协作能力得到明显的提高。

## 6.3下一步工作

首先计划对代码进行优化，剔除冗余代码，增加注释提升代码可读性和可维护性。其次，学习更加优秀的压缩代码对软件性能进行优化，在保证压缩性能的同时，降低时间复杂度，使软件获得更优良的时间性能。另外，对界面进一步优化，增加诸如文件预览等更加实用的功能。

## 6.4个人设计小结

### 6.4.1白千一设计小结

本次课题完成过程中，最提升能力的地方不是代码的编写，是开始阶段对题目的整体架构。拿到课题题目后，没有马上开始着手编写代码，而是进行了大量思考，对项目整体的框架进行了完善的设计，将不同的功能封装成不同的类，尽量降低类和类之间额的耦合性，使整个项目的条理十分清晰。代码开始编写的很晚，但是完成的很快。

另外，在此次任务的完成过程中，学习到了Git和GitHub的相关知识，提升了团队协同合作的能力。

不足之处在于，没有查阅足够的资料，软件的编写仅仅围绕一个哈夫曼算法，其实本可以通过结合其他压缩算法，将压缩软件做的更好。最后产生结合其他压缩算法的想法时已经来不及了。

### 6.4.2金忠信设计小结

在本次课题完成过程中，通过和组长之间反复讨论，我对哈夫曼算法的理解更为深刻。在代码编写过程中，各种类之间的逻辑关系和框架构造，以及对文件的操作，以及STL中vector数组的使用巩固了我对C++和数据结构的认识。

不足的地方在于，通过这次实验认识到自己C++编程方面的不足，还需加强。曾尝试过基于LZ77理论和游程编码将压缩算法进行优化，但是由于时间限制，未能良好地将这种优化方法与哈夫曼算法结合。

1. 附录

## A课题任务分工

### A-1课题程序设计分工

**课题程序设计分工**

|  |  |  |  |
| --- | --- | --- | --- |
| 学号 | 姓名 | 程序设计函数原型、类 | 功能说明 |
| 20194701 | 白千一 | 1. 字母表类（alphaTable） 2. 哈弗曼树（HuffmanTree） 3. Bit转化类（BitDeal） 4. 文件读写类（FileRW） 5. 压缩窗口类（ComDialo） | 1. 读取文件建立字母表，向文件写入头数据 2. 读取字母表建立哈夫曼树表、编码表 3. 位运算改变字符型数据（char）的位序列 4. 文件编码译码，压缩解压缩，进行各种文件的转化 5. 压缩界面的实现 |
| 20194744 | 金忠信 | 1. 结点类（HNode） 2. 字母类（alpha） 3. 哈弗曼树（HuffmanTree 4. 主窗口类（MainWindow） 5. 解压窗口类（DecomDialog） | 1. 构成哈夫曼树表，以及相关运算符的重载。 2. 构成字母表，以及相关运算符的重载 3. 读取字母表建立哈夫曼树表、编码表 4. 主窗口界面的实现 5. 解压界面的实现 |

### A-2课题报告分工

**课题报告分工**

|  |  |  |
| --- | --- | --- |
| 章节 | 内容 | 完成人 |
| 1 课题概述 | 1.1 课题任务  1.2 课题原理  1.3 相关知识 | 白千一 |
| 2 需求分析 | 2.1 课题调研  2.2 用户需求分析 | 金忠信 |
| 3 方案设计 | 3.1 总体功能设计  3.2 数据结构设计  3.3 函数原型设计  3.4 输入输出设计  3.5 主算法设计  3.6 用户界面设计 | 白千一  金忠信 |
| 4 方案实现 | 4.1 开发环境与工具  4.2 程序设计关键技术  4.3 个人设计实现（按组员分工）  4.3.1白千一个人设计实现  4.3.2金忠信个人设计实现 | 白千一  金忠信 |
| 5 测试与调试 | 5.1 个人测试（按组员分工）  5.1.1白千一测试  5.1.2金忠信测试  5.2 组装与系统测试 | 白千一  金忠信 |
| 6 课题总结 | 6.1 课题评价  6.2 团队协作  6.3 下一步工作  6.4 个人设计心得（按组员分工）  6.4.1白千一设计小结  6.4.2金忠信设计小结 | 白千一  金忠信 |

## B课题设计文档

### B-1源程序代码（见附录项目文件夹）

### B-2工程与可执行文件（见附录项目文件夹）