中国科学技术大学计算机学院
《计算机系统概论实验报告》

实验题目：huffuman压缩

学生姓名：叶子昂

学生学号：PB20020586

完成时间：2021年12月29日

实验要求

0.1 基本要求

> 基于 Huffman 编码实现一个压缩器和解压缩器（其中 Huffman 编码以字节作为统计和编码的基本符号单元），使其可以对任意的文件进行压缩和解压缩操作。针对编译生成的程序，要求压缩和解压缩部分可以分别独立运行。具体要求为：
>
> 每次运行程序时，用户可以指定只压缩/只解压缩指定路径的文件。实现的时候不限制与用户的交互方式，可供参考的方式包括但不限于
>
> - 根据命令行参数指定功能（压缩/解压缩）和输入/输出文件路径
> - GUI 界面
> - 运行程序后由用户交互输入指定功能和路径
>
> 【CAUTION!】不被允许的交互方式： 通过修改源代码指定功能和文件路径
>
> 压缩时不需要指定解压文件的目标路径，解压缩时不需要指定压缩前原文件的路径，压缩后的文件可以换到另一个位置再做解压缩

0.2 完成的要求

- 基本要求(通过命令行完成)
- 0.5到四字节为单位的压缩，2到16叉树任意组合的压缩

0.2.1 输入：./main.exe [option] [arguments]

```
            cout << "-h : print out help information" << endl;
            cout << "-f : the path for the input file" << endl; //-f 后应接文件路径及文件名
            cout << "-o : the path for the output file" << endl;//-o 接输出路径文件名可不带
            cout << "-c : compress the input file" << endl;       // 压缩时应指定-c(compress)后接                                                    分叉数
            cout << "-b : the number of minimum half byre"<<endl; //后接字节单位相对于0.5的倍数
            cout << "-d : decode the input file" << endl;          //解压时应指定-d(decode)
```

示例:(具体示例见代码测试部分)
4叉树以3字节(即6个半字节)为单位压缩

> ./main.exe -c **4** -b **6** -f input**path**+inputfilename -o **outputpath**

解压

> ./main.exe -d -f inputpath+name.yza -o outputpath

0.2.2 输出：程序运行进度,Huffman编码,输出文件目录(解压时),原文件大小压缩文件大小(压缩时)

设计思路

0.3 命令行控制

- 命令行信息获取

```
pair<bool, string> getCmdOption(char **begin, char **end, const string &option)
{//获取cmd信息
      char **itr = find(begin, end, option);
      if (itr != end && ++itr != end) {
            return make_pair(true, *itr);
     }
      return make_pair(false, "");
}

bool cmdOptionExists(char **begin, char **end, const string &option) {//检测cmd指
令是否存在
      return find(begin, end, option) != end;
}
```

- 命令行信息处理

```
       auto input_info = getCmdOption(argv, argv + argc, "-f");
      auto output_info = getCmdOption(argv, argv + argc, "-o");
      auto cmode = getCmdOption(argv, argv + argc, "-c");
      auto bmode = getCmdOption(argv, argv + argc, "-b");
```

0.4 代码结构

0.4.1 压缩

- 记录文件名,压缩方式于headinformation
- 读取文件,按处理单元统计出现频率.
- 根据统计频率构造huffman树
- 根据huffman树进行Huffman编码
- 写入压缩文件,先写入headinformation,再写入Huffman树,最后根据Huffman编码编码写入压缩文件,最后可能不足一字节,补0后写入,最后有不足一单元的直接写入,同时记录有效bits个数.写入文件尾部.

0.4.2 解压

- 首先读入文件名,原压缩方式,有效字节数.
- 重构Huffman树
- 根据重构的Huffman树和有效字节数限定的范围解压文件

关键代码分析

0.5 字节处理宏定义

```
#define check_byte(byte,  bit) (((byte) & (1 << ((bit)))) != 0)//检查byte的第bit位为1还是0
#define set_byte1(byte, bit) ((byte) |= (1<<bit))//设置byte的第bit位为1
#define set_byte0(byte, bit) ((byte) &= (~(1 << bit)))//设置byte的第bit位为0
#define dihead_byte(res, byte) ((res = (byte>>4)&0xF))//获取byte的前四位即前半字节
#define ditail_byte(res, byte) ((res = byte&0xF))//获取byte的后四位即后半字节
#define sethead_byte(byte, data) ((byte = (data<<4)&0xF0))//将byte的前半字节置为data
#define settail_byte(byte, data) ((byte = (byte|0xF)&(data|0xF0)))//将byte的后半字节置为data
```

0.6 关键数据结构

- 统计查找表

```
typedef class FLNode{//节点
public:
   unsigned char* data;
   long freq;
   string code;//Huffman编码
   FLNode* next;
   FLNode(){data = nullptr; freq =0;code="";next=nullptr;}
   FLNode(unsigned char* da, long fr,string co)
{data=da;freq=fr;code=co;next=NULL;}
}* flnode;

class findlist{//查找统计表
public:
   flnode head;
   int size;
   findlist(){head=new FLNode(nullptr,0,"");size=0;}
   bool empty(){if(!head->next)return 1;else return 0;}
   flnode find(unsigned char* elem);//查找结点
   Status ListInsert(unsigned char*& elem);//没有插入,有则frequency+1;统计
};
```

- 优先队列(链式)

```
class priorityquene{//优先队列
public:
	HuffmanTree head;
	HuffmanTree tail;
	int size;

	priorityquene(){head=new Node();tail=new Node();size=0;}
	bool empty(){if(!head->next)return 1;else return 0;}
	HTNode* top();
	HTNode* pop();
	Status push(HTNode* node);//按freq小在前大在后插入
};
```

0.7 压缩

0.7.1 读文件并统计频率

- 设计思路
  - 由于压缩单位为0.5到4字节,为考虑通用性,读入时统一按字节读满两倍的最小单位,然后拆分为两个单位
  - 最后文件可能不足最小单位,个人的处理方法为读到最后一个完整单位,剩下不满一个单位的不统计,在一般情况下,这样造成的影响可以忽略不计.
- 实现代码

```
Status Readingangcouting(ifstream& ist){//统计频率
	unsigned char buf;
	// unsigned char** data = new unsigned char * [2];
	// data[0] = new unsigned char [byunit];
	// data[1] = new unsigned char [byunit];
	unsigned char *buffer = new unsigned char [byunit];
	long filesize=0;
	int count=0;
	cout<<"reading the file..."<<endl;
	ist.seekg(0L,ios::end);
	filesize= ist.tellg()-ist.tellg()%byunit;//计算出文件含多少最小单位
	headinformation.laststraghtbit=ist.tellg()%byunit;//记录未被统计字节个数
	ist.clear();
	ist.seekg(0L,ios::beg);
	for(int i=0;i<filesize ;i++){
		ist.read(reinterpret_cast<char *>(&buf),sizeof(char));
		buffer[count]=buf;
		count++;
		if(count>=byunit){//读满两倍的最小单位
			unsigned char** data = new unsigned char * [2];
			data[0] = new unsigned char [byunit];
			data[1] = new unsigned char [byunit];
```

```
                int j=0;
                unsigned char temp;
                for(int i=0; i<byunit;i++){//拆分
                      dihead_byte(temp,buffer[i]);
                      if(j<byunit){
                            data[0][j]=temp;j++;
                      }else{
                            data[1][j-byunit]=temp;j++;
                      }
                      ditail_byte(temp,buffer[i]);
                      if(j<byunit){
                            data[0][j]=temp;j++;
                      }else{
                            data[1][j-byunit]=temp;j++;
                      }
                }
                count=0;
                freq.ListInsert(data[0]);//插入统计表
                freq.ListInsert(data[1]);//插入统计表
          }
      }
       return OK;
}
```

0.7.2 创建Huffman树

- 设计思路
  - 将节点入优先队,对多叉树需要计算补成满多叉树
  - 使用优先队列可以方便的得到freq小的前几位,从而构造Huffman树
- 代码实现

```
Status CreateHuffmanTree(HuffmanTree& tree){
      int addi=0;
      int weight=0;
      priorityquene Q;
      if(freq.empty()){cout<<"empty list"<<endl;exit(ERROR);}
      flnode p;
      for (p=freq.head->next;p;p=p->next) {
            HTNode* node=make_node(p->data,p->freq,nullptr);
            Q.push(node);
      }
      addi=(Q.size-1)%(forks-1);
      if(addi)addi = forks-1-addi;//计算需补全节点数
      for(int i=0;i<addi;i++){//对多叉节点补全
            HTNode* node=make_node(nullptr,0,nullptr);
            Q.push(node);
      }
      cout<<"building the huffmantree..."<<endl;
```

```
        while(Q.size!=1){//构造Huffman树
            HTNode** child=new HTNode* [forks];
            for(int i=0;i<forks;i++){
                child[i]=Q.top();
                Q.pop();
                weight+=child[i]->weight;
            }
            HuffmanTree father=make_node(nullptr,weight,child);
            Q.push(father);
        }
        tree=Q.top();
        return OK;
    }
```

0.7.3 Huffman编码

按递归先序遍历Huffman树遍历到叶子节点后对其编码

值得注意的是由于可能为多叉树,编码需要如下特殊处理

```
const string num_c[16]={//2叉树取一位,3到4叉树取两位,5到8叉树取三位,9到16叉树取四位
                "0000",
                "0001",
                "0010",
                "0011",
                "0100",
                "0101",
                "0110",
                "0111",
                "1000",
                "1001",
                "1010",
                "1011",
                "1100",
                "1101",
                "1110",
                "1111",
};
inline string numtostr(int i,int length){//取指定编码
    string str;
    int place = 4-length;
    str=num_c[i];
    str.erase(0,place);
    return str;
}
```

代码实现:

```
Status CreateHuffmanCode(HuffmanTree& tree,string str){//先序遍历树进行huffman编码
      if(tree==NULL)return OK;
      if(check_leaf(tree)){
            if(str==""){//仅有一个元素根节点
                  flnode p=freq.find(tree->data);
                  if(!p)return OK;
                  p->code="1";
            }
            else {
                  flnode p=freq.find(tree->data);
                  if(!p)return OK;
                  p->code=str;
            }
      }
      else {
            for(int i=0; i<forks ;i++){
                  // if(tree->child[i]->data=='~')continue;
                  string temp=numtostr(i,codebit);
                  CreateHuffmanCode(tree->child[i],str+temp);
            }
      }
      return OK;
}
```

0.7.4 压缩

- 压缩顺序
  - 写入原文件名,压缩方式,直接写入的字节个数
  - 写入Huffman树
  - 写入编码后文件,最后不足最小单元的直接写入
  - 写入不足一字节的有效bit位数
- 代码实现

```
Status write_compressfile(ifstream& ist,ofstream& ost,HuffmanTree& root){
      cout<<"compressing the file..."<<endl;
      unsigned char buf;
      unsigned char tempdata;
      int bit=0;
      ist.clear();
      ist.seekg(0,ist.beg);
      cout<<"writing headinformation..."<<endl;
      ost<<headinformation.filename<<"\n"<<forks<<"\n"<<byunit<<"\n"
<<headinformation.laststraghtbit<<endl;//写入文件信息
      writehuffmantree(root,ost);//写入huffmantree
      ost<<endl;
      cout<<"encoding the file..."<<endl;
      unsigned char** data = new unsigned char * [2];
      data[0] = new unsigned char [byunit];
```

```
data[1] = new unsigned char [byunit];
unsigned char* buffer = new unsigned char [byunit];
long filesize=0;
int count=0;
ist.seekg(0L,ios::end);
filesize= ist.tellg()-ist.tellg()%byunit;
ist.clear();
ist.seekg(0L,ios::beg);
for(int i=0;i<filesize ;i++){
      ist.read(reinterpret_cast<char *>(&buf),sizeof(char));
      buffer[count]=buf;
      count++;
      if(count==byunit){
            int j=0;
            unsigned char temp=0;
            for(int i=0; i<byunit;i++){
                  dihead_byte(temp,buffer[i]);
                  if(j<byunit){
                        data[0][j]=temp;j++;
                  }else{
                        data[1][j-byunit]=temp;j++;
                  }
                  ditail_byte(temp,buffer[i]);
                  if(j<byunit){
                        data[0][j]=temp;j++;
                  }else{
                        data[1][j-byunit]=temp;j++;
                  }
            }
            flnode p=freq.find(data[0]);
            for(int i=0;i<p->code.size();i++){
                  if(p->code[i]=='0'){
                        set_byte0(tempdata,bit);
                  }
                  else{
                        set_byte1(tempdata,bit);
                  }
                  bit++;
                  if(bit>=8){
                        bit=0;
                        ost.write((char*)&tempdata,sizeof(unsigned char));
                  }
            }
            p=freq.find(data[1]);
            for(int i=0;i<p->code.size();i++){
                  if(p->code[i]=='0'){
                        set_byte0(tempdata,bit);
                  }
```

```
                        else{
                              set_byte1(tempdata,bit);
                        }
                        bit++;
                        if(bit>=8){
                              bit=0;
                              ost.write((char*)&tempdata,sizeof(unsigned char));
                        }
                  }
                   count=0;
            }
      }
       if(bit){//最后压缩字节
             lastvaildbit=bit;
             headinformation.lastvaildbit=bit;
             ost.write((char*)&tempdata,sizeof(unsigned char));
       }else {tempdata = 0; ost.write((char*)&tempdata,sizeof(unsigned char));}
       char left='\0';
       while(ist>>left)
             ost<<left;
       ost<<lastvaildbit;
       //统计信息
       long long beforecompress=0;
       long long aftercompress=0;
       ist.clear();
       ist.seekg(0L,ist.end);
       beforecompress=ist.tellg();
       aftercompress=ost.tellp();
       //
       ist.close();
       ost.close();
       cout<<"finished compression"<<endl;
       cout<<"the size of the original file is "<<beforecompress<<endl;
       cout<<"the size of the compressed file is "<<aftercompress<<endl;
       return OK;
}
```

0.8 解压缩

0.8.1 重建Huffman树

- 设计思路
  - 根据获取的源文件压缩方式创建对应的Huffman树
  - 原Huffman先序写入,重构时按先序重构即可.
- 代码实现

```
Status ReCreateHuffman(ifstream& ist,HuffmanTree& root){//重新构建huffumantree
      unsigned char* data= new unsigned char [byunit];
      unsigned char temp;
```

```
        char label;//标识
        for(int i=0; i<byunit ;i++){
              ist.read(reinterpret_cast<char *>(&temp),sizeof(unsigned char));
              data[i]=temp;
        }
        ist.read(&label,sizeof(unsigned char));
        if(label=='1'){//叶子节点
              root=make_node(data,0,nullptr);
              return OK;
        }
        else{//中间节点
              root=new HTNode();
              for(int i=0;i<forks ;i++){//多叉递归构造
                    ReCreateHuffman(ist,root->child[i]);
              }
              return OK;
        }
        return OK;
  }
```

0.8.2 解码文件

- 设计思路
  - 读入压缩文件头信息包括文件名,多少叉压缩的,最小压缩单元等
  - 重建Huffman树
  - 根据重建好的Huffman树解码文件
- 代码实现

```
  Status HuffmanDecoder(HuffmanTree& tree,ifstream& ist, ofstream& ost){//解压缩
        int i=0;
        char uselesschar;
        unsigned char temp;
        // char tempwrite;
        int bit=0;
        long long filesize;
        long long cuplace=0;
        HuffmanTree p;
        string filename = output_filename;

  //ist文件处理及头信息获取和处理*****************************

        ist.seekg(0L,ist.end);
        filesize=ist.tellg();
        ist.clear();
        ist.seekg(0L,ist.beg);
        cout<<"Geting head information..."<<endl;
        ist.seekg(-1L,ist.end);
        ist>>headinformation.lastvaildbit;
```

```
        lastvaildbit=headinformation.lastvaildbit-48;
        ist.clear();
        ist.seekg(0L,ist.beg);

    ist>>headinformation.filename>>forks>>byunit>>headinformation.laststraghtbit;
        laststraghtbit=headinformation.laststraghtbit;
        if (forks == 2);
        else if(forks>=3&&forks<=4)codebit=2;
        else if(forks>=5&&forks<=8) codebit=3;
        else if(forks>=9&&forks<=16) codebit=4;
        else{
              cout << "overrange of forks" << endl;
              exit(ERROR);
       }
        if(filename.empty())
              filename=headinformation.filename;
        else
              filename=output_filename+headinformation.filename;
        ost=fileput(filename);
        ist.read(&uselesschar,sizeof(char));
        cout<<"Rebuilding HuffmanTree..."<<endl;
        ReCreateHuffman(ist,tree);
        ist.read(&uselesschar,sizeof(char));
        p=tree;

//读写译码****************************************
        HuffmanTree pdata[2]={nullptr,nullptr};
        int count=0;
        int place = 0;
        int getplace = 0;
        cout<<"decoding the file..."<<endl;
        unsigned char* buffer= new unsigned char [byunit];
        ist.read((char*)&temp,sizeof(unsigned char));
        cuplace=ist.tellg();
        while(!ist.eof()){
              if(check_leaf(p)){
                    pdata[count]=p;count++;
                    if(count==2){//等待两个单元合并
                          mergebits(buffer,pdata[0]->data,pdata[1]->data);
                          ost.write(reinterpret_cast<char *>
(buffer),byunit*sizeof(char));
                          test+=(char*)buffer;count=0;
                    }
                    p=tree;
                    if(cuplace>=(filesize-1-laststraghtbit)&&bit>=lastvaildbit){
                          break;
                    }
              }
```

```
                // place = forkdir(temp,bit);
                if(check_byte(temp,bit)){
                        place += pow(2,codebit-getplace-1);
                }
                getplace++;
                if(getplace == codebit){
                        getplace =0;
                        p=p->child[place];
                        place = 0;
                }
                bit++;
                if(bit>=8){
                        bit=0;
                        ist.read((char*)&temp,sizeof(unsigned char));
                        cuplace=ist.tellg();
                }
        }
        char tempdata;
        for(int i=0;i<laststraghtbit;i++){ist>>tempdata;ost<<tempdata;}
        ist.close();
        ost.close();
        cout<<"success in decoding!\nthe output file is "<<filename<<endl;
        return OK;
}
```

调试分析

0.9 算法的时空复杂度分析

> 以文件大小为问题规模，压缩字节单元为1和分叉数为2。

0.9.1 时间复杂度

> (统计表和Huffman叶子节点最多有256个元素)涉及树和查找类操作只需O(1)与文件大小无关

- 读文件O(n)(统计表最多有256个元素)
- 创建Huffman树为O(1)(节点最多有256个)
- 压缩需要逐字节读逐位分析为O(8n)亦为O(n)
- 重构Huffman树为O(1)
- 解压缩需要逐字节读逐位分析为O(8n)亦为O(n)

0.9.2 空间复杂度

> (统计表和Huffman叶子节点最多有256个元素)它们的存储只需O(1)与文件大小无关

- 程序运行过程中只需暂存原文件和输出文件故为O(n).

0.10 遇到的问题

- 编码或译码一两个单元后直接写入造成大量重复操作增加压缩和解压缩的时间，或许可以通过设立一定大小的缓冲区，当填满整个缓冲区后一次性写入。
- 为了代码通用性考虑，对任意字节单元统一采用拆成半字节存储（如一字节为单位压缩时，存储时存为两个半字节）这样无论是整字节还是半字节都需经过字节处理花费相当大的时间。
- 在字节单元选取较大时如4字节会导致统计表十分巨大最多有256x256x256x256约4×10^9个元素，导致即使压缩非常小的文件也耗时巨大，仅适合压缩文本。

代码测试

输入（对picture.jpg 以3叉树0.5字节进行压缩）

```
PS D:\wh030917\Documents\GitHub\huffmancompress\huffman> .\main.exe -c 3 -b 1 -f .\picture.jpg -o D:\wh030917\Documents\GitHub\huffmancompress\
```

部分输出（vscode终端简省问题导致Huffman编码展示不全）

```
reading the file...
building the huffmantree...
creating the huffmancode...

__________________________
|str    |code   |
|
        |0100   |
encoding the file...
finished compression
the size of the original file is 1666848
the size of the compressed file is 2225931
```

输入：（解压上述压缩文件）

```
PS D:\wh030917\Documents\GitHub\huffmancompress\huffman> .\main.exe -d -f ..\.yza -o D:\wh030917\Documents\GitHub\huffmancompress\
```

输出：

```
Geting head information...
Rebuilding HuffmanTree...
decoding the file...
success in decoding!
the output file is D:\wh030917\Documents\GitHub\huffmancompress\picture.jpg
```

检查解压图可以正常打开且与原文件一致。

实验总结

- 通过本次实验我在实践中了解了Huffman树的结构，实现与应用。在解决问题的过程中了解了优先队列的基本结构，并能够简单的实现链式优先队列。
- 本次实验设计文件处理的部分较多，附加实验让我掌握了半字节的处理方法。
- 实验存在大文件压缩慢，对除文本以外文件压缩率低甚至比原文件大的情况，需要综合其他方法优化。

附录

- report
- main.cpp　主程序，包含各种压缩和解压缩的函数主
- huffman.hpp　一些Huffman树相关数据结构的声明和方法服务于main中函数
- file.hpp　文件处理相关基本操作，及查找统计表的定义和方法
- user.h　通用宏定义和库函数包含，同时包含简单的inline函数