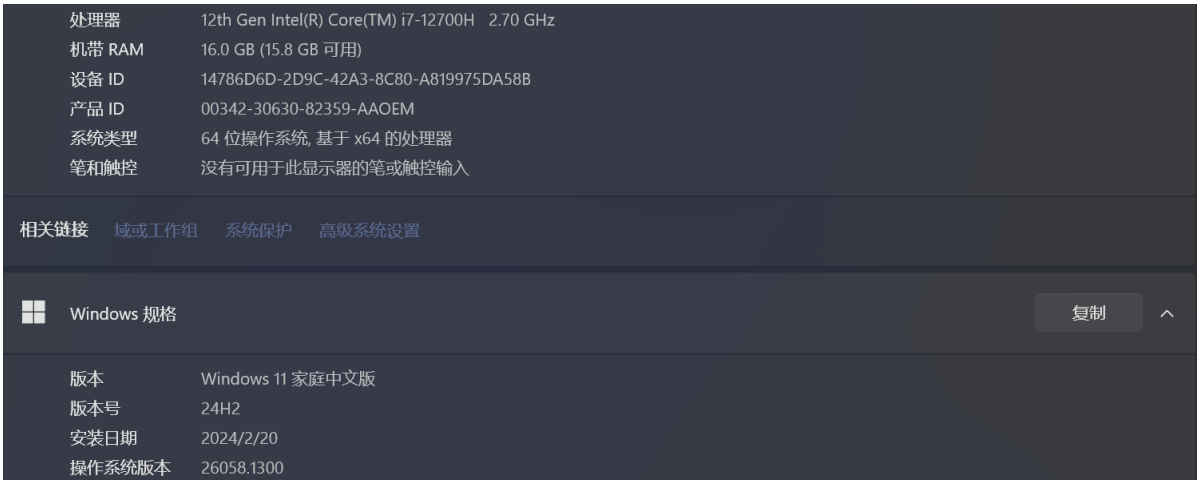
# 522031910747+李若彬+lab0


522031910747+李若彬+lab0
- 实验环境
- 哈夫曼树的实现
  - 哈夫曼树类
  - 哈夫曼树的建树过程
  - 功能函数
- 实验设计
  - 三组不同的文本
  - 分析


## 实验环境




## 哈夫曼树的实现

### 哈夫曼树类

- 私有成员：
  - 首先需要定义节点类

```
struct Node
{
  std::string   data;//储存的字符
  int       frequency;//字符出现频率
  Node      *left;//左儿子
  Node      *right;//右儿子
  Node( std::string data, int frequency, Node * left, Node * right ) : data( data ), frequency( frequency ), left( left ), right( right )//节点构造函数
  std::string getMinData() const {//获取节点最小字典序
      if ( this->left == nullptr && this->right == nullptr )
      {
          return(data);//如果是叶子节点，直接返回节点的数据
      }
      std::string minData = this->data;//初始化最小字典序为当前节点的数据
      if ( this->left != nullptr )
```

```
        {
            std::string leftMinData = this->left->getMinData();//递归获取左子树的最小字典序
            if ( leftMinData < minData )
            {
                minData = leftMinData;//更新最小字典序
            }
        }
        if ( this->right != nullptr )
        {
            std::string rightMinData = this->right->getMinData();//递归获取右子树的最小字典序
            if ( rightMinData < minData )
            {
                minData = rightMinData;//更新最小字典序
            }
        }
        return(minData);
    }
};
```

- 比较函数（用于后续哈夫曼树构造的优先级队列）

```
static bool compare( const Node* a, const Node* b )
{
    if ( a->frequency == b->frequency )
    {
        std::string minDataA    = a->getMinData();//获取树a中所有节点的最小字典序
        std::string minDataB    = b->getMinData();// 获取树b中所有节点的最小字典序
        return(minDataA < minDataB);//比较最小字典序
    }
    return(a->frequency < b->frequency);
}
```

- 一个节点类的指针

```
Node *ptr;
```

- 公有成员（`handout`已给出的`skeleton code`）
  - `Option`
  - 构造函数
  - 编码函数

```
enum class Option
{
    SingleChar,
    MultiChar
};
hfTree( const std::string & text, const Option op );
std::map<std::string, std::string> getCodingTable();
```

# 哈夫曼树的建树过程

- 构造函数

```
hfTree::hfTree(const std::string &text, const Option op)
{
    // TODO: Your code here
    std::deque<Node *> forest;//创建一个双端队列forest用于存储节点
    if (op == Option::SingleChar)//单字符压缩
    {
        std::map<char, int> charCount;//用一个频率表map来存储字符&频率
        for (char c : text) {
            charCount[c]++;
        }
        for (auto it = charCount.begin(); it != charCount.end(); it++)
{//使用迭代器将map对应的键值对存入队列中
            Node *node = new Node(std::string(1, it->first), it-
>second, nullptr, nullptr);
            forest.push_back(node);
        }
        int j = forest.size();
        for (int i = 0; i < j - 1; i++) {//构造哈夫曼树
            std::sort(forest.begin(), forest.end(), compare);//始终保持队
列中头两个节点最小
            ptr = new Node(forest[0]->data + forest[1]->data,
forest[0]->frequency + forest[1]->frequency, forest[0], forest[1]);
            forest.pop_front();
            forest.pop_front();
            forest.push_back(ptr);
        }
        ptr = forest.front();//返回哈夫曼树的根节点
    }
    else if (op == Option::MultiChar)//多字符压缩
    {
        std::map<std::string, int> charCount;
        for (int i = 0; i < text.size(); i ++) {
            std::string temp = text.substr(i, 1);
            charCount[temp] = 0;
        }
        std::map<std::string, int> tempCount;//用于存储两个字符的键值对
        for (int i = 0; i < text.size() - 1; i ++) {
            std::string temp = text.substr(i, 2);
            tempCount[temp]++;
        }
        std::vector<std::pair<std::string, int>> topCombinations;//将两
个字符的键值对转换为vector类型便于后续比较
        for (auto it = tempCount.begin(); it != tempCount.end(); it++)
{
            topCombinations.push_back(*it);
        }
        //对两个字符的键值对排序，频率较大的在前面，如果频率一样则字典序较小的在前
面
        std::sort(topCombinations.begin(), topCombinations.end(), []
(const auto& a, const auto& b) {
            if (a.second == b.second) {
```

```
                return a.first < b.first;
            }
            return a.second > b.second;
        });
        for (int i = 0; i < 3; i++) {//将频率最大的三个（不足三个的按全部）两个字符的键值对插入到map
            if (i >= topCombinations.size()) break;
            std::string temp = topCombinations[i].first;
            charCount[temp] = 0;
        }
        for (int i = 0; i < text.size(); i ++) {//用map中的键扫描text内容，并更新其值
            std::string currentTwoChars = text.substr(i, 2);//首先先匹配两个字符的键
            if (charCount.find(currentTwoChars) != charCount.end()) {
                charCount[currentTwoChars]++;
                i++;
            }
            else {//两个字符没有匹配成功，匹配单个字符
                std::string currentOneChar = text.substr(i, 1);
                if (charCount.find(currentOneChar) != charCount.end())
{
                    charCount[currentOneChar]++;//匹配到单个字符的字符串，计数加一
                }
            }
        }
        auto it = charCount.begin();
        while (it != charCount.end()) {//删除频率表map中值为0对应的键值对
            if (it->second == 0) {
                it = charCount.erase(it);
            } else {
                ++it;
            }
        }
        for (auto it = charCount.begin(); it != charCount.end(); it++)
{
            Node *node = new Node(it->first, it->second, nullptr,
nullptr);
            forest.push_back(node);
        }
        int j = forest.size();
        for (int i = 0; i < j - 1; i++) {
            std::sort(forest.begin(), forest.end(), compare);
            ptr = new Node(forest[0]->data + forest[1]->data,
forest[0]->frequency + forest[1]->frequency, forest[0], forest[1]);
            forest.pop_front();
            forest.pop_front();
            forest.push_back(ptr);
        }
        ptr = forest.front();
    }
}
```

- 编码

```cpp
std::map<std::string, std::string> hfTree::getCodingTable()
{
    // TODO: Your code here
    if (ptr != nullptr) {//哈夫曼树根节点不为空
        std::map<std::string, std::string> codingTable;//用一个map来储存键值对
        std::string code;//霍夫曼编码
        std::queue<std::pair<Node *, std::string>> q;//用一个双端队列来存节点和编码，其每个元素都是一个pair
        q.push(std::make_pair(ptr, ""));//压入根节点
        while (!q.empty()) {
            Node *node = q.front().first;//获取队列中的第一个节点
            code = q.front().second;//获取节点当前编码
            q.pop();
            if (node->left == nullptr && node->right == nullptr) {//叶子节点
                codingTable[node->data] = code;//更新map的键值对
            }
            if (node->left != nullptr) {//有左孩子，将左孩子压入队列，编码加0
                q.push(std::make_pair(node->left, code + "0"));
            }
            if (node->right != nullptr) {//有右孩子，将右孩子压入队列，编码加1
                q.push(std::make_pair(node->right, code + "1"));
            }
        }
        return codingTable;
    }
    return std::map<std::string, std::string>();
}
```

## 功能函数

- 读取待压缩编码的文件内容

```cpp
std::string parseText( const std::string &input )
{
    std::string content;
    // TODO: Your code here
    std::ifstream file( input );
    if ( file.is_open() )
    {
        char c;
        while ( file.get( c ) )
        {
            content += c;
        }
        file.close();
    }
    return(content);
}
```

- 将`data`中的内容输出到`output`指定的文件

```cpp
void output( const std::string &output, const std::string &data )
{
    // TODO: Your code here
    std::string extension = output.substr( output.find_last_of( "." ) + 1 );//查找输出文件的扩展名
    if ( extension == "huffzip" )//输出到zip文件
    {
        std::ofstream outputFile( output, std::ios::binary );//以二进制形式打开文件
        if ( outputFile.is_open() )
        {
            std::string byteString;
            for ( int i = 0; i < data.size(); i += 8 )//将编码数据按照8位一组转换为字节并写入文件
            {
                std::string byte        = data.substr( i, 8 );
                uint8_t     byteValue   = 0;
                for ( int j = 0; j < 8; j++ )//将8位编码转换为一个字节的数值
                {
                    byteValue <<= 1;
                    if ( byte[j] == '1' )
                        byteValue |= 1;
                }
                //byteValue 是一个 uint8_t 类型的变量，它占用一个字节的内存空间。reinterpret_cast 将其地址转换为 const char* 类型的指针，以便将字节数据写入文件。write 函数根据指针和字节数将数据写入文件
                outputFile.write( reinterpret_cast<const char *>(&byteValue), sizeof(byteValue) );
            }
            outputFile.close();
        }else  {
            std::cerr << "Failed to open output file" << std::endl;
        }

    }else  {//正常输出字符串
        std::ofstream file( output );
        if ( file.is_open() )
        {
            file.write( data.c_str(), data.size() );
            file.close();
        }
    }
}
```

- 将编码表转为字符串

```
std::string codingTable2String( const std::map<std::string, std::string> &codingTable )
{
    std::string result = "";
    // TODO: Your code here
    for ( auto it = codingTable.begin(); it != codingTable.end(); it++ )
    {
        result += it->first + " " + it->second + "\n";//原符号+空格+对应编码+换行符
    }
    return(result);
}
```

- 读取编码表

```
void loadCodingTable( const std::string &input, std::map<std::string, std::string> &codingTable )
{
    // TODO: Your code here
    std::ifstream file( input );
    if ( file.is_open() )
    {
        char buffer[2];//一次读取两个字符
        while ( file.read( buffer, sizeof(buffer) ) )
        {
            if ( file.peek() == '0' || file.peek() == '1' )//下一个字符为数字，说明为单个字符，多取了一个字符
            {
                std::string key( buffer, 1 );//只取buffer的第一个字符作为key
                std::string value;
                std::string plus( buffer, 2 );
                value += plus;
                std::getline( file, value );//读取0或1后的所有字符作为value
                codingTable[key] = value;//更新键值对
            }else if ( file.peek() == ' ' )//下一个字符为空格，说明正好取两个字符
            {
                file.ignore( 1 );//忽略空格
                std::string key( buffer, sizeof(buffer) );//这两个所取得字符即为键
                std::string value;
                std::getline( file, value );//读取空格后的所有字符作为value
                codingTable[key] = value;//更新键值对
            }
        }
        file.close();
    }
}
```

- 压缩

```
std::string compress( const std::map<std::string, std::string> &codingTable, const std::string &text )
```

```cpp
{
    std::string result;
    // TODO: Your code here
    std::string bitstream;//用于存储有效比特流
    std::string currentCode;
    for ( size_t i = 0; i < text.size(); ++i )
    {

        //优先比较两个字符
        if ( i + 1 < text.size() )
        {
            std::string doubleChar = text.substr( i, 2 );
            if ( codingTable.count( doubleChar ) > 0 )
            {
                bitstream += codingTable.at( doubleChar );
                i += 1;
                continue;
            }
        }
        //如果两个字符的匹配失败，那么就匹配一个字符
        std::string singleChar = text.substr( i, 1 );
        if ( codingTable.count( singleChar ) > 0 )
        {
            bitstream += codingTable.at( singleChar );
        }
    }
    uint64_t effectiveBits = bitstream.size();//存储比特流的有效位
    for ( int i = 0; i < 8; ++i )
    {
        result += static_cast<char>( (effectiveBits >> (i * 8) ) & 0xFF
);
    }
    while ( bitstream.size() % 8 != 0 )//将比特流补0使得成为字节流
    {
        bitstream += '0';
    }
    std::string ans = "";
    std::stack<int> binaryEffectiveBits;//将比特流的有效位数转换为二进制
    while ( effectiveBits > 0 )
    {
        binaryEffectiveBits.push( effectiveBits % 2 );
        effectiveBits /= 2;
    }
    std::string binaryStr = "";
    while ( !binaryEffectiveBits.empty() )
    {
        binaryStr += std::to_string( binaryEffectiveBits.top() );
        binaryEffectiveBits.pop();
    }
    while ( binaryStr.size() != 64 )//将二进制的有效位数补0使得其有64位
    {
        binaryStr = "0" + binaryStr;
    }
    std::string reverse = "";
    int     jj  = binaryStr.size() / 8;
    for ( int i = 0; i < jj; i++ )//以小端的方式表示64位的二进制下的有效位数
```

```
        {
            reverse += binaryStr.substr( binaryStr.length() - 8 );
            binaryStr.erase( binaryStr.length() - 8 );
        }
        ans = reverse + bitstream;
        result = ans;
        return result;
    }
```

# 实验设计

- 三组不同的文本（长度有较为显著的差异， 但三个文件的大小之和不超过 1 MB）
- 比较两种压缩方式在这些文本上压缩效果的差异

## 三组不同的文本

1. 

   文本大小：

   大小: 8.83 KB (9,045 字节)

   `sin.huffzip`大小：

   大小: 5.03 KB (5,152 字节)

   `mul.huffzip`大小：

   大小: 5.02 KB (5,144 字节)

   `sin`压缩率：$5152 \div 9045 = 56.9596\%$

   `mul`压缩率：$5144 \div 9045 = 56.8712\%$

   `mul`相较于`sin`方式压缩提升率：$(5152 - 5144) \div 5152 = 0.1553\%$

2. 

   文本大小：

   大小: 328 KB (336,394 字节)

   `sin.huffzip`大小：

   大小: 204 KB (209,038 字节)

   `mul.huffzip`大小：

   大小: 201 KB (206,556 字节)

   `sin`压缩率：$209038 \div 336394 = 62.1408\%$

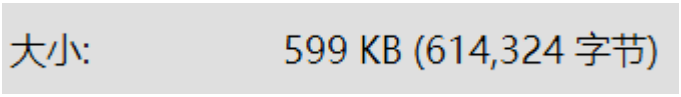
   `mul`压缩率：$206556 \div 336394 = 64.4030\%$

   `mul`相较于`sin`方式压缩提升率：$(209038 - 206556) \div 209038 = 1.1873\%$

3. 

   文本大小：


`sin.huffzip`大小：




`mul.huffzip`大小：


`sin`压缩率：$614324 \div 822364 = 74.7022\%$

`mul`压缩率：$572807 \div 822364 = 69.6537\%$

`mul`相较于`sin`方式压缩提升率：$(614324 - 572807) \div 614324 = 6.7582\%$

## 分析

- `sin`和`mul`两种压缩方式都在一定程度上减少了原文件的大小，压缩率大约在50%以上，且原文件越大，压缩率也相应增加，压缩效果对应降低
- `mul`比`sin`方式在各种原文件大小的情况下压缩率都更小，压缩效果都更好，且随着原文件大小的增加，`mul`比`sin`在压缩方面的提升越来越大
- 对于更优的压缩策略：当原始文件大小较大时，应当更多地使用连续两个字符参与编码，例如设置参与的数量与原始文本大小正相关，因为在大量的文本中会有大量的重复字符串，这样可以更高效率地压缩并且能达到更低地压缩率，同时还可以适当地引入一些更长的连续字符参与编码，这样会大大降低编码时的资源使用，并压缩后的文件大小也会显著降低。