

# VITTER算法：一种改进的自适应霍夫曼编码方案

## ——适用于任何文件格式的纯C实现

高振明 20222005253 软工四班

*In the "Three Little Pigs," the wolf had to huff and huff to blow those houses down. He should have just **adaptive huffed**.*

## 一、人员和分工

- 所有工作：高振明 20222005253 软工四班

## 二、问题描述

### 要求（选题2）

1. 使用Huffman编码设计一个压缩软件
2. 能对输入的任何类型的文件进行哈夫曼编码，产生编码后的文件——压缩文件；
3. 能对输入的压缩文件进行译码，生成压缩前的文件——解压文件；
4. 要求编码、译码效率尽可能地高；（选做）
5. 撰写实验报告

## 三、工具/准备工作

### 涉及到的数据结构知识

- Huffman Coding
- Tree Structure
  - 树的操作
  - 带权树

### 使用的C++IDE

我并未使用任何IDE，采用Visual Studio Code代码编辑器 + clang + make在macOS上编译运行。本项目使用**纯C代码**编写，不调用任何std库函数，但使用了一些POSIX API. 经过测试在类Unix（macOS, Linux, BSD）系统上都能够编译运行。但由于MSVC不自带POSIX接口，在Windows下需要通过minGW环境进行编译（**VC++或者VS都不能编译**）。此外，众所周知只有Windows还没有全面使用UTF-8编码，这意味着在默认中文终端环境下（GBK编码），运行本程序将会乱码（因为在UTF-8环境下编写）。基于此，无论是源代码还是程序反馈我都使用英文编写以避免不必要的麻烦。

## 四、分析与实现

(具体实现：我在代码中写了较为详细的注释，请务必参看注释)

### 想法：用什么实现

用于我们使用自适应Huffman编码（下文介绍），相比于传统Huffman，我们不需要任何高级数据结构，字符串不需修改，只用增加，使用 `char**` (C-Style String) 最快，没有面向对象的需求，因此整个解决方案使用纯C语言实现，最为便捷和快速。

### 基础分析：如何读入压缩/待压缩文件和输出解压/压缩文件

在实际实现( `huff.c` )中，有这样几个函数：

```
1 int AHEDGetInputBufferNextBit(FILE* file) //读取单个比特
2 int AHEDGetInputBufferChar(FILE* file) //读取一个字节，调用数次上述的函数获取多个比特组成一个字节
3 int AHEDPutBit2Buffer(FILE* file, int value) //将一个比特放入输出缓冲区
4 int AHEDPutChar2Buffer(FILE* file, unsigned char c) //多次调用上述函数构成一个字节并放入输出缓冲区
```

负责读入/输出每一个字节。这几个函数的底层都是 `stdio` 中的 `getc()` 和 `putc()`。得益于这些按字节读入/输出的函数，我们能够计算压缩文件和原文件的大小，并计算出其压缩率。

### 难点分析：如何使HUFFMAN编码适用于任何类型的文件？

Huffman编码是一个经典的Entropy-Based的压缩算法，关于它的各种语言的实现层出不穷。但Huffman编码一般用于文本的压缩，若需要压缩各类文件（i.e. 以二进制方式压缩），则需要在实现上作出改进。

符号表是Huffman编码中重要的概念，我们需要用符号表来统计每个符号的出现频数（权重），对于一个文本文件来说，其符号表就是文本中出现的所有不同字符的集合，而这些字符也是组成这个文件的最小单位（不考虑比特）。然而二进制文件则一般不使用字符来统计，二进制文件的最小单位是**字节**。我们可以依此类推，只要计算出所有字节的种类即可。显然，根据定义有  $1B = 8b$ ，利用乘法原理我们可以轻松得到字节共有  $2^8 = 256$  个（0x00 ~ 0xFF）。——相对的，英语字母表只有52个。不妨设第*i*个符号为 $a_i$ ，对应的权重为 $w_i$ ，对 $n \in (1, 256]$ 可以构造函数

$$f: A \rightarrow W \text{ where } A = \{a_1, \dots, a_n\}, W = \{w_1, \dots, w_n\} \quad (1)$$

这是一个从符号映射到权重的单射函数，通过这个函数我们就完成了符号权重的统计。

### 难点创新：如何改进原版HUFFMAN编码

静态Huffman编码是一个2-Pass编码，依赖于原文件各个符号的频率序列。也就是说为了构建一棵Huffman树（构造函数 $f$ ）需要读入一遍原文件，在这棵Huffman树上编码的时后又要读一遍，这不仅效率较低，同时还有一个隐性要求：我们要预先知道这个文件的结尾在哪，才能开始编码。考虑直播时的情况，我们是不可能知道直播流的结尾（EOF）在哪，就没法实时对之编码。除非对于每一个新增的字节，我们都重新跑一遍整个（2-Pass的）流程，这显然**慢的离谱**，效率上划不来。下面介绍一种改进的动态算法。

**Adaptive Huffman Coding**是由Faller, Gallager, Knuth三人提出来并由Vitter进行改进的，现时一般称为Vitter算法。这个算法能够实时编码，填补了上述的不足。**Vitter算法中的每一个结点都有一个序号，称为Order**。本次实训采用这个算法实现程序。Vitter算法基于重要的Sibling Property:

### Lemma Sibling Property 兄弟性质

一棵具有 $p$ 个带权叶结点树是Huffman树当且仅当其满足 ( $N_i$ 表示第 $i$ 个结点) :

1. 前 $p$ 个带权叶子结点都是非负的
2. 任意的带权内结点的权值都等于其子孙权值的和
3. 结点可以按权值大小构成一个不下降的序列 $\mathbf{s.t.} \forall p(1 \leq j \leq p-1 \rightarrow N_{2j-1}, N_{2j} \text{是兄弟关系})$ , 且他们的父节点序号比他们自身高

换句话说, 对于一个以结点序号构成的不下降序列 $y_1, y_2, \dots, y_{2n-1}$ , 若有对应的权值序列 $x_1, x_2, \dots, x_{2n-1}$ 满足

$$x_1 \leq x_2 \leq \dots \leq x_{2n-1} \quad (2)$$

$\iff$  这棵树是Huffman树

也就是说, 任意的树, 只要满足上述引理即成为一棵Huffman Tree, 可以遍历编码。我们只要能找到一种动态的、自适应的算法构建出一棵Huffman树就可以达成目标了。这就是Vitter算法的思想: **保证Sibling Property不被破坏**。

Vitter算法较之传统Huffman算法多了几个**特点**:

- 结点具有序号
- 存在空结点 (也称为控制结点, Not Yet Transferred, NYT, 标志其之后还有符号没有传输)。在代码中, 我使用 `zero_node` 来指代这个结点。最开始的Huffman树只具有一个结点, 即空结点。
- 具有相同权值和相同类型 (叶子结点; 内结点) 的结点集合称为Block

联系上面说过的256个字节, 显然整棵树最多512个结点, 其中叶子结点256个、内结点255个, 加上一个NYT结点, 即可得。

## 性能

对于一条长度为 $t$ 并包含 $n$ 个符号信息, 令 $T$ 为总共传输的bits, 那么有 (Vitter, 1987)

$$S - n + 1 \leq T \leq 2S + t - 2n + 1 \text{ where } S = \sum_j w_j l_j \quad (3)$$

此处的 $S$ 指的是Static Huffman编码的性能 (WPL)。Vitter接着指出在最坏情况下, 传输的bit会比传统Huffman多一个。

## 更新树

**Vitter算法依据读入的符号动态更新Huffman Tree**。这是传统Huffman算法做不到的。每读入一个新的符号, 就需要往树中加入这个符号, 为了保证Sibling Property, 我们需要对树进行操作, 称为Update. 更新树的操作在代码中是函数 `AHEDActualizeTree(tree_array, actual_node)`, 可以用流程图清晰的表示出这个函数即更新操作的逻辑:

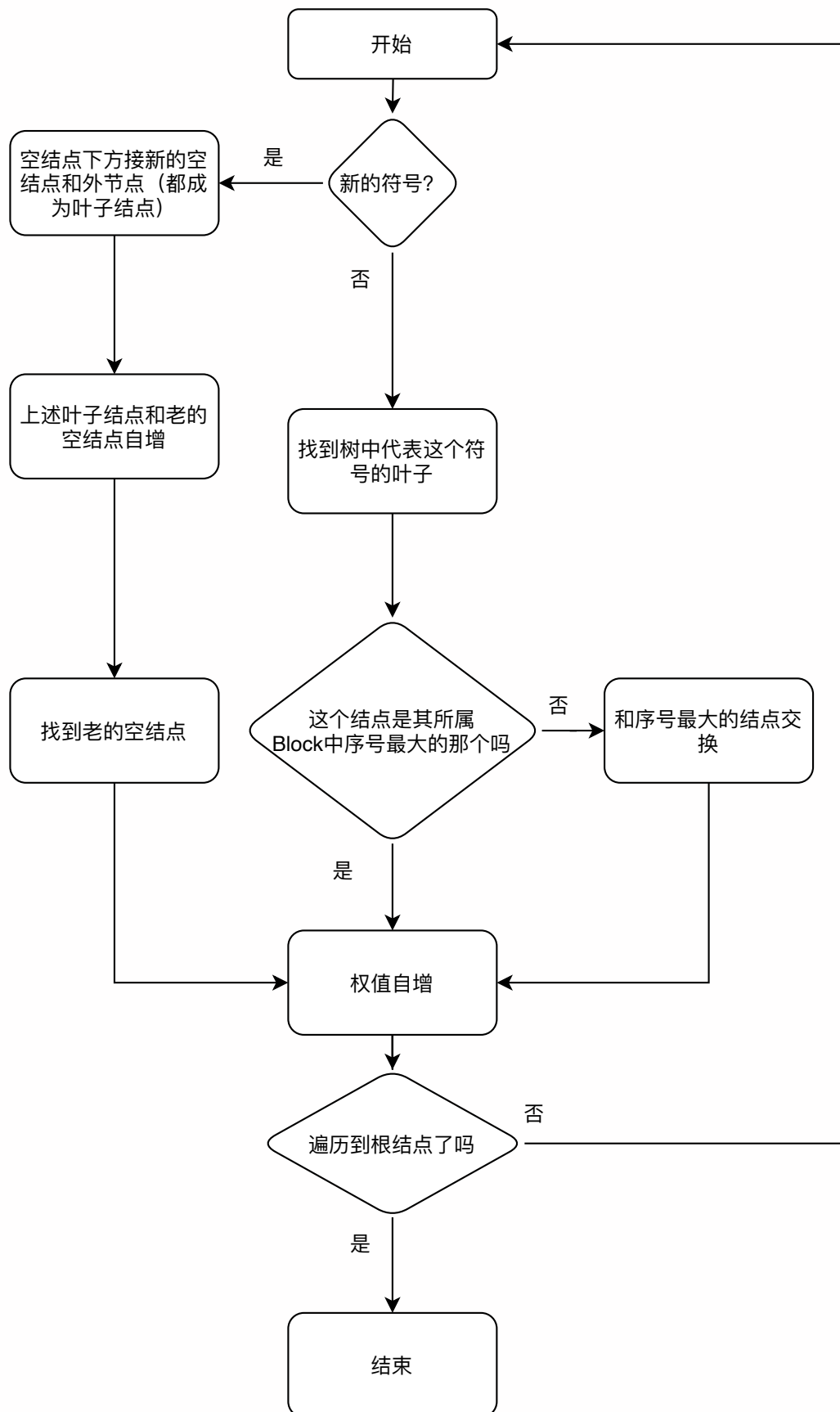


Fig 1. Update Procedure

通过不断的更新树，我们保证了Sibling Property，这也是为什么最后能进行Huffman编码的原因，这是Vitter算法的最大优势。

用pseudocode可以写成如下形式：

```

1  begin
2      Initialize
3      readSymbol(X)
4      while (X≠EOF) do
5          begin
6              if (first_read_of(X)) then
7                  begin
8                      output(code of ZERO)
9                      output(code of node X)
10                     create new node U with next node ZERO and new node X
11                     update_tree(U);
12                 end
13             else
14                 begin
15                     output(code of node X)
16                     update_tree(node X)
17                 end
18             readSymbol(X)
19         end
20     end
21
22     procedure update_tree(U)
23     begin
24         while (U≠root) do
25             begin
26                 b := block of nodes preceding node U
27                 if ((U is leaf) and (b is block of inner nodes) and (value U = value b))
28                     or ((U is inner node) and (b is block of leaves) and (value U = value b - 1)) then
29                     begin
30                         slide U in front of block b
31                         increment value of U
32                         if (U is leaf) then
33                             U := parent(U)
34                         else
35                             U := parent (U before slide)
36                     end
37                 else
38                     begin
39                         increment value U
40                         U := parent(U)
41                     end
42             end
43         increment value U
44     end

```

显然，NYT结点在整个树中只会出现一次。而且必然是叶子结点。

需要注意的是，一次更新操作也比较繁。整体来看在编码速度上有可能不如Static Huffman Coding.

对于树的动态更新就是AHC的主要内容，有了上面的介绍，我们能具体列表对比一下两种编码的差异：

## Huffman Code

## Adaptive Huffman Code

静态，需要提前知道文件中各个符号的统计数据

自适应，根据实时读入的文件动态更新树

在我测试的几个文件（bmp, docx, 可执行程序, 纯文本）中

右边的压缩率略胜于左边 ( $\Delta < 0.02$ )

一般利用优先队列(e.g. C++的 `priority_queue`)和链表

只需要简单的数据结构：树和向量。

## 编码

正如上面所说，Vitter算法的特点在于动态调整Huffman Tree。因此无论是编码还是解码都相对简单，类似传统Huffman。但是不同于传统算法，AHC中有两种编码。

- NYT编码。上面说过NYT符号表示之后还有符号没被传输，NYT结点作为这个树的叶子结点，同样会被传输，这样解码器才知道。

### Definition NYT Code

$$\text{NYT Code} = \{\text{从根结点出发到对应NYT结点的路径}\} \quad (4)$$

上面仅仅是NYT一个符号的编码方案。由于AHC是一个实时算法，因此编码器和解码器（发送方和接收方）必须要提前约定好所有符号的编码方案（Scheme），Vitter指出，如果令 $n$ 为符号表的大小（**i.e.**  $n = |A|$ ），

$$\text{find } (b, r) \text{ s.t. } \begin{cases} n = 2^b + r \\ 0 \leq r < 2^b \end{cases} \quad (5)$$

其中， $b$  表示编码长度（按比特数计算）； $r$  表示「余数」，指不能被长度为 $b$ 编码的数据大小。满足(5)的一个 $(b, r)$ 就是发送方和接收方需要提前约定好的参数。

- 定长编码。

### Definition Fixed Code

1. 对 $A$ 中的符号 $a_i$ ，如果满足 $0 \leq i < 2r$ ，那么 $a_i$ 的值（编码）是 $(i - 1)_{10}$ ，长度 $b + 1$
2. 否则， $a_i$ 表示为 $(i - r - 1)_{10}$ ，长度为 $b$ 。

列表可得

长度	编码范围 (Base10)	值 (Base10)
$b + 1\text{bit}$	$[1, 2r]$	$i - 1$
$b \quad \text{bit}$	$[2r + 1, n]$	$i - r - 1$

严格说来，这并不是定长编码，考虑  $n = 5, b = 2, r = 1$ ，很明显对值  $[1, 2]$ ，长度为3；对值  $[3, 5]$ ，长度为2。不符合定长编码的定义，但是近似：最多只有两种长度的编码。**考虑我们的情况**，也就是  $n = 256$ ，显然  $(8, 0)$  符合定义。因为  $r = 0$ ，只需考虑第二种情况，亦即我们能用长度为8的编码编码整个符号表。这样，我们就厘清了AHC的具体编码方案。

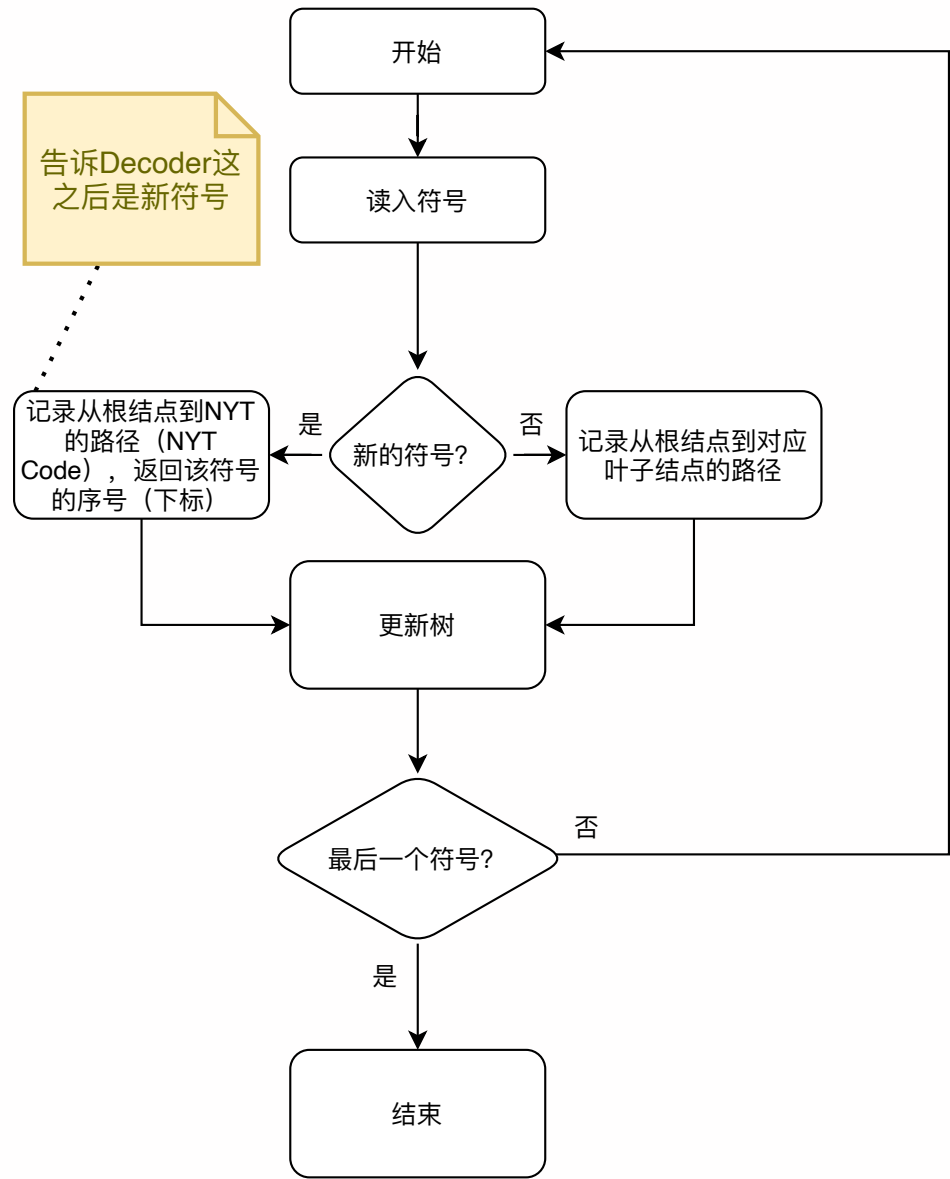


Fig 2. Encoding Procedure

解码

解码过程和编码过程极其相似。我们读入压缩过的文件并动态构建Huffman树，需要重点指出的是，对于同一个原文件，编码和解码过程构造出的Huffman树应该是一致的，否则实现就出现了问题，造成解码文件和原文件不一样。我们可以通过计算两者的md5 hash比较得出结果（我实际也是这么做的）。

用流程图可以表示如下：

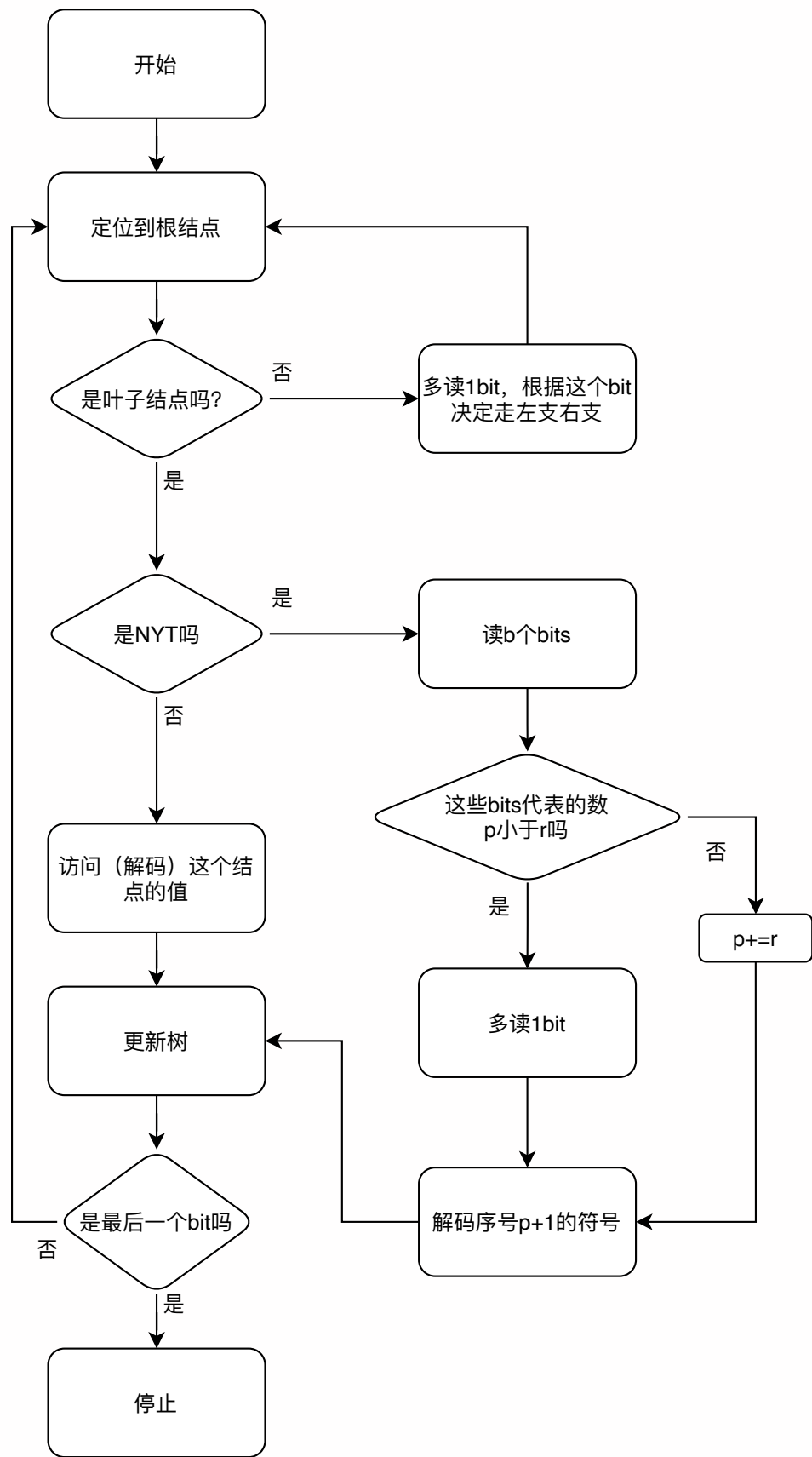


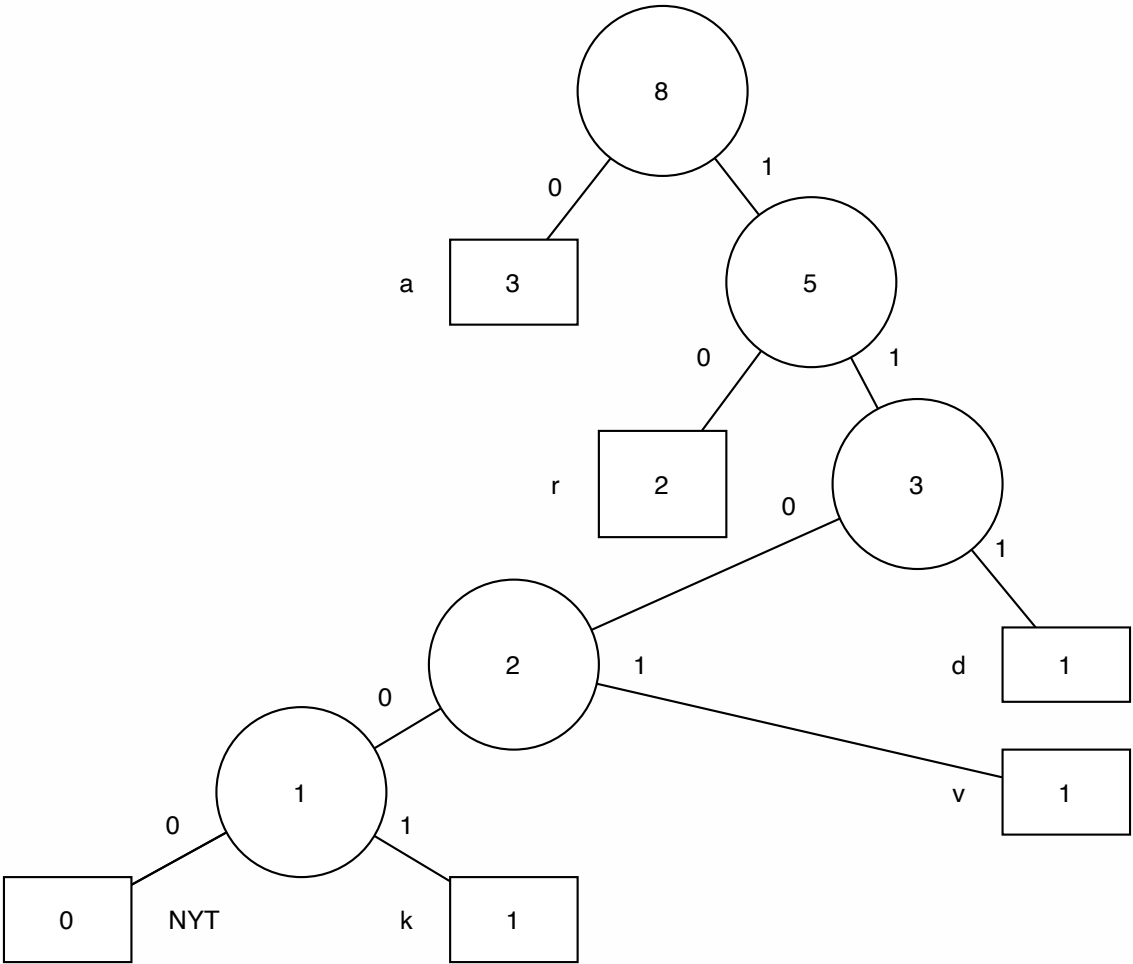


Fig 3. Decoding Procedure

e.g. 考虑  $n = 26, b = 4, r = 10$  (英文字母) 和序列

```
00000101000100000110001011010110001010
```

建树可得



按照上述流程解码可得

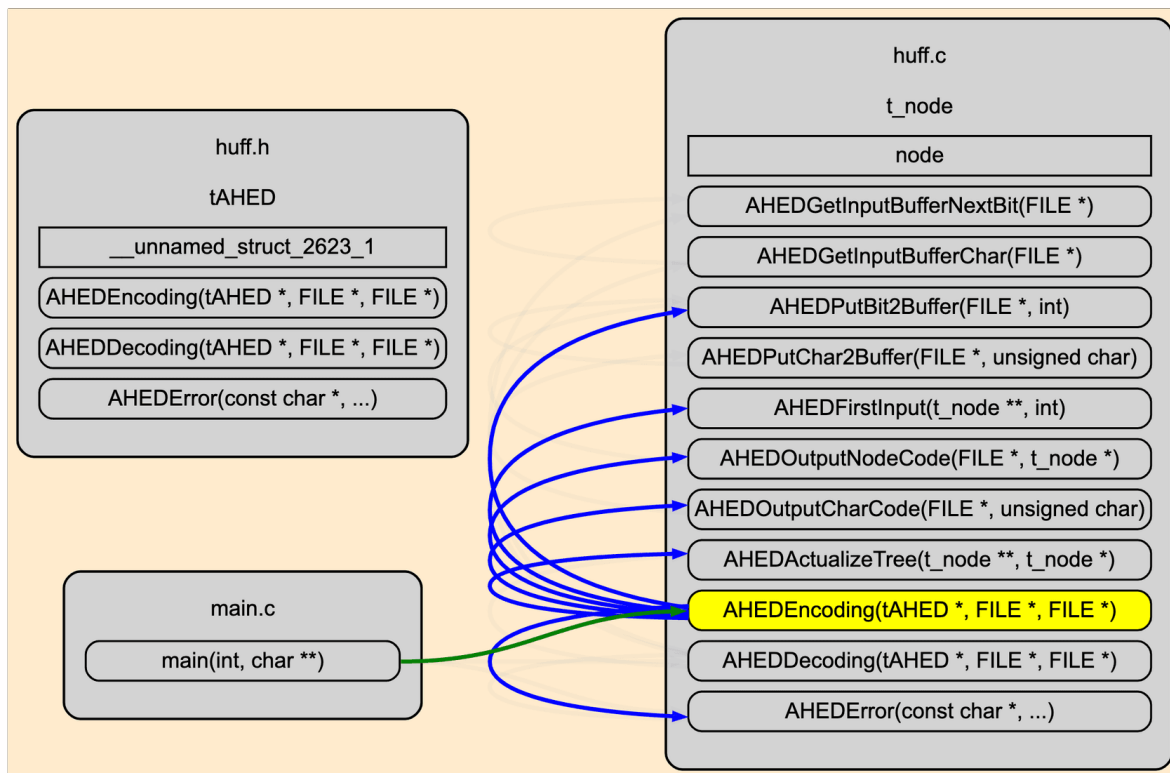
1	00000	1	0	10001	00	00011	000	1011	0	10	1100	01010
2	a	a	NYT	r	NYT	d	NYT	v	a	r	NYT	k

这样，我们就介绍完了完整的Vitter算法。

### 函数调用关系

以下列出本项目两个主要的函数和所有的函数调用关系图

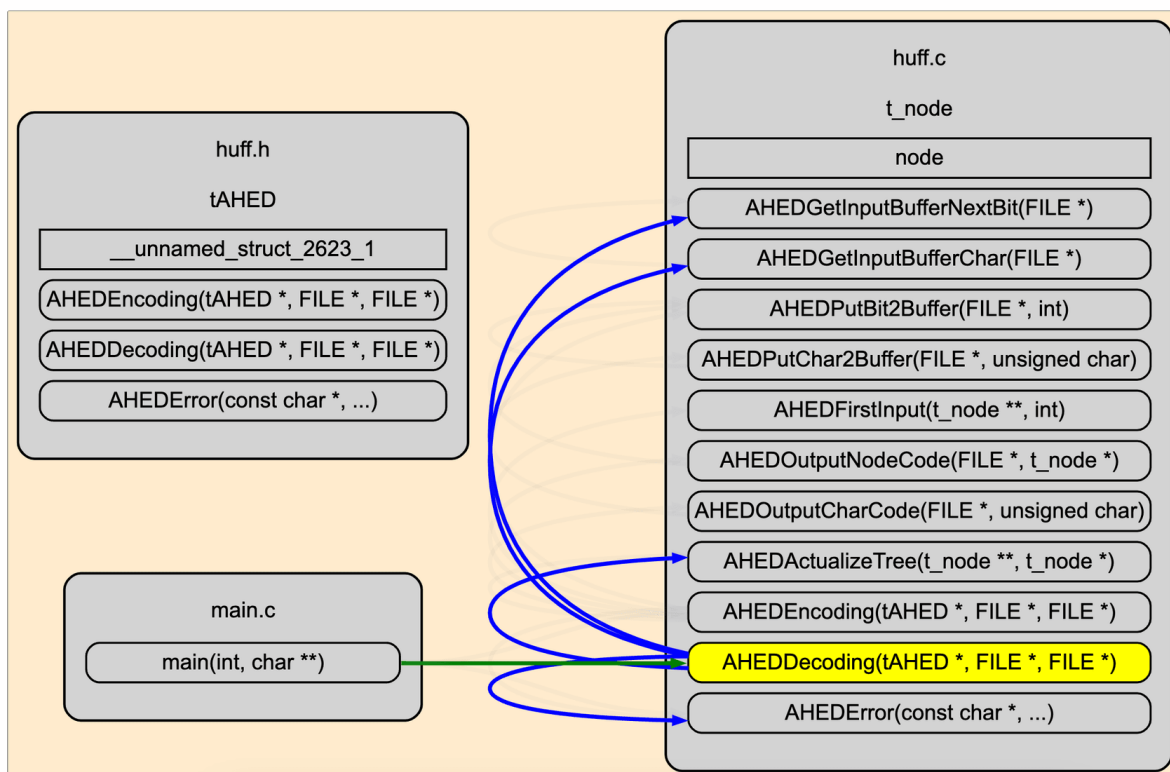
AHEDEncoding() ——负责编码的函数



上面介绍过算法的工作原理了，因此作为编码函数，需要调用

- `AHEDActualizeTree()` 来更新树
  - 其中又需要调用 `AHEDFirstInput()` 来判断符号是否出现过
- `AHEDOutput*()` , `AHEDPutBit` 负责输出编码
- `AHEDError()` 为了符合程序设计规范，增加错误处理功能，程序出错时提供有效报错信息。

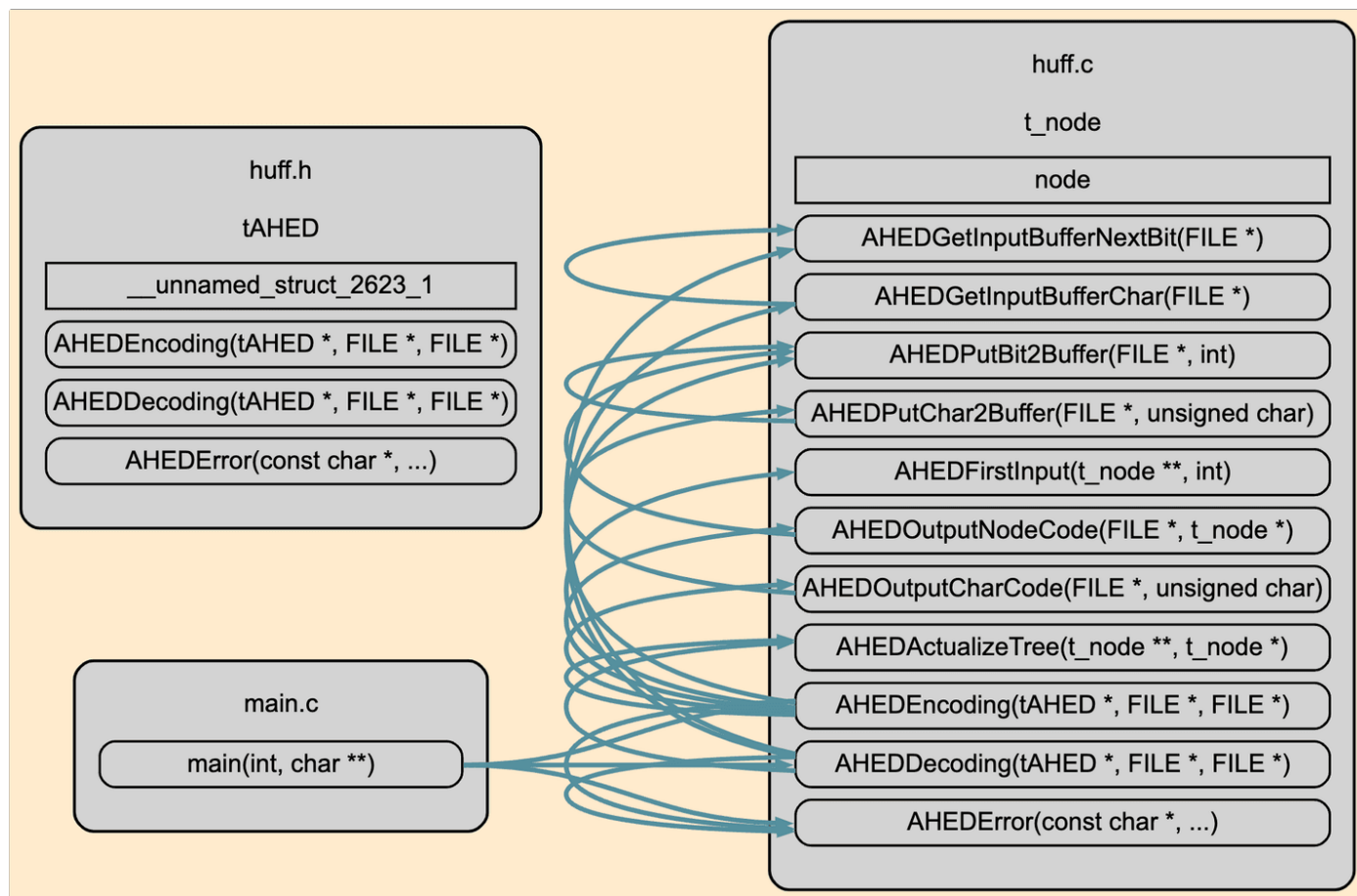
`AHEDDecoding()` ——负责解码的函数



- `AHEDActualizeTree()` 来更新树

- 其中又需要调用 `AHEDFirstInput()` 来判断符号是否出现过
- `AHEDGetInput*()` 负责读入编码
- `AHEDError()` 为了符合程序设计规范，增加错误处理功能，程序出错时提供有效报错信息。

整体调用图：



## 五、测试与结论

### 程序使用说明

本程序为命令程序，参数列表为 `-h | -q | -c | -x [-i input_file] [-o output_file] [-l log_file]`，其中：

- `-h`，help 输出帮助信息

```
1 | USAGE: main -h | -q | -c | -x [-i input_file] [-o output_file] [-l log_file]
```

- `-q` 开关, quiet 不显示编解码信息到 `stdin`，如果不指定输出（默认 `stdout`）则直接输出编解码的结果。
- `[-i input_file] [-o output_file]`，指定输入（默认：`stdin`）输出（默认：`stdout`）文件。输入输出参数常搭配 `-c -x` 开关使用。
  - `-c`，create 压缩开关
  - `-x`，extract 解压开关
- `[-l log_file]`，log 指定日志文件

e.g. 下面提供了一些常见的使用例子

编码 `OGtext.txt` 并pipe到程序来压缩再pipe来解压，最后重定向到`text.txt`，用`time`计时

```
1 | time cat OGtext.txt | ./main -c -q | ./main -x > text.txt
2 | # 所需时间
3 | ./main -c -q 0.77s user 0.00s system 14% cpu 5.272 total
4 | ./main -x -q 0.64s user 0.17s system 14% cpu 5.581 total
```

可以看出，对于一个6M，12万行的文本编解码都不到1s，性能不错。

编码 `OGpic.bmp` 并输出压缩文件到 `encOGpic.bmp`，保存日志到 `enc.log`，安静模式

```
1 | ./main -c -i OGpic.bmp -o encOGpic.bmp -l enc.log -q
```

通过tarball打包多个文件并使用本程序压缩（`-` 在Unix下指 `stdout`）

```
1 | tar -cf - ${SOURCE_FILES} | ./main -c -o ${OUTPUT_DIR}
```

程序的一个典型输出如下：

```
1 | =====Encoding===== # 表示编码过程
2 | Encode from OGdoc.docx to encOGdoc.docx
3 |
4 | Time elapsed: 120.752000ms during ENCODING process # 统计操作所耗时间
5 | n_Symbols=256, Encoded Size=118268 B, Decoded Size=118479 B, Compressed ratio = 99.821909%
6 | # 分别表示{符号数量}{编码后大小}{编码前大小}{压缩率}
7 | =====Completed=====
```

## 可执行文件

目录下有两个可执行文件：

- `main` 是在本地电脑（ARM64/macOS）上构建的，仅仅适用于苹果M芯片的Mac。
- `main.exe` 是通过CI/CD(Github Action)在平台（AMD64/Windows）上通过MSYS2工具链构建的，经过测试能够在其他电脑上运行。

## 测试运行

为了确保实现的正确性，我准备了四种类型的文件并对之进行编码，解码测试，通过计算并比较其原文件和解码文件的md5来确定文件是否相同。若相同则说明我们的无损压缩实现是没有问题的。

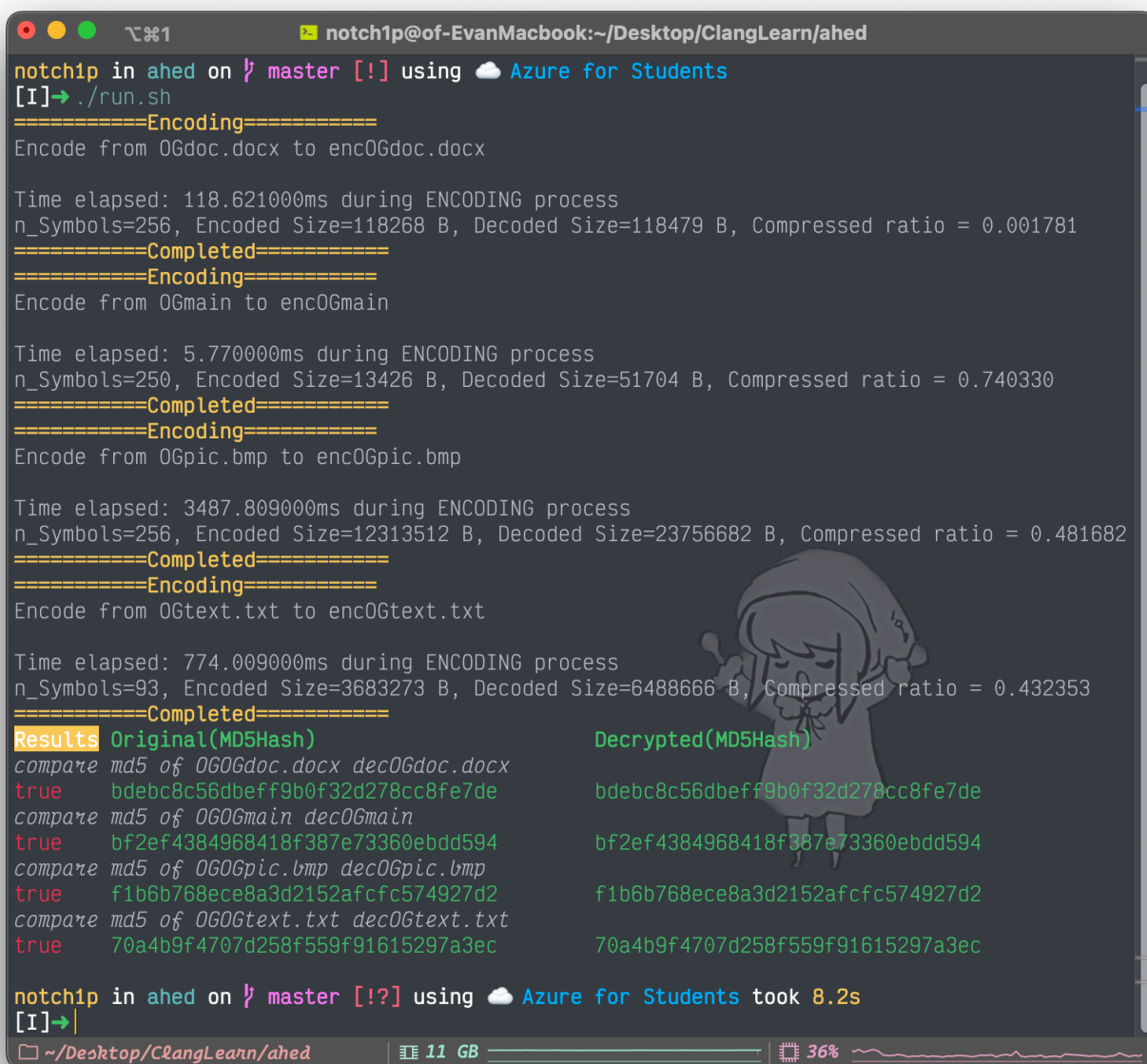
据此，我写了脚本 `run.sh` 来自动化这一过程：

```

1  for i in OG{doc.docx,main,pic.bmp,text.txt}; do
2      ./main -c -i $i -o enc$i
3      ./main -x -i enc$i -o dec$i -q
4  done
5  echo
6  "${On_IYellow}Results${ENDCOLOR}\t${BIGreen}Original(MD5Hash)\t\tDecrypted(MD5Hash)${ENDCOLOR}"
7  for i in OG{doc.docx,main,pic.bmp,text.txt}; do
8      echo "\033[3mcompare md5 of OG$i dec$i\033[0m"
9      md51="$(md5 -q $i)"
10     md52="$(md5 -q dec$i)"
11     [ "$md51" = "$md52" ] && echo
12     "${RED}true${ENDCOLOR}\t${IGreen}${md51}\t${md52}${ENDCOLOR}"
13 done

```

运行本脚本，可以得到输出如图所示：



```

notch1p in ahed on  master [!] using  Azure for Students
[I]→ ./run.sh
=====Encoding=====
Encode from OGdoc.docx to encOGdoc.docx

Time elapsed: 118.621000ms during ENCODING process
n_Symbols=256, Encoded Size=118268 B, Decoded Size=118479 B, Compressed ratio = 0.001781
=====Completed=====
=====Encoding=====
Encode from OGmain to encOGmain

Time elapsed: 5.770000ms during ENCODING process
n_Symbols=250, Encoded Size=13426 B, Decoded Size=51704 B, Compressed ratio = 0.740330
=====Completed=====
=====Encoding=====
Encode from OGpic.bmp to encOGpic.bmp

Time elapsed: 3487.809000ms during ENCODING process
n_Symbols=256, Encoded Size=12313512 B, Decoded Size=23756682 B, Compressed ratio = 0.481682
=====Completed=====
=====Encoding=====
Encode from OGtext.txt to encOGtext.txt

Time elapsed: 774.009000ms during ENCODING process
n_Symbols=93, Encoded Size=3683273 B, Decoded Size=6488666 B, Compressed ratio = 0.432353
=====Completed=====
Results  Original(MD5Hash)                Decrypted(MD5Hash)
compare md5 of OGOGdoc.docx decOGdoc.docx
true    bdebc8c56dbeff9b0f32d278cc8fe7de    bdebc8c56dbeff9b0f32d278cc8fe7de
compare md5 of OGOGmain decOGmain
true    bf2ef4384968418f387e73360ebdd594        bf2ef4384968418f387e73360ebdd594
compare md5 of OGOGpic.bmp decOGpic.bmp
true    f1b6b768ece8a3d2152afcfc574927d2        f1b6b768ece8a3d2152afcfc574927d2
compare md5 of OGOGtext.txt decOGtext.txt
true    70a4b9f4707d258f559f91615297a3ec        70a4b9f4707d258f559f91615297a3ec

notch1p in ahed on  master [!] using  Azure for Students took 8.2s
[I]→

```

可以看出，解码后程序的md5和原文件相同，这证明了实现不存在重大错误。我们不妨将上述结果制成表格，如下所示：

(定义**压缩率** $r = 1 - \frac{S_{enc}}{S_{OG}}$  (越高越好)，其中 $S$ 表示文件大小 (in bytes)； $enc$ 表示编码文件； $OG$ 表示原文件； $n$ 表示符号数； $t_{enc}/t_{dec}$ 表示编码/解码时间 (ms) )

文件 (类型)	OGtext.txt (文本)	OGpic.bmp (位图)	OGmain (程序)	OGdoc.docx (Word)
$S_{enc}$	3683273	12313512	13426	118268
$S_{OG}$	6488666	23756682	51704	118479
$n$	93	256	250	256
$t_{enc}/t_{dec}$	748/586	3374/2440	5/4	120/62
$r \cdot 100\%$	43.2353%	48.1682% <sup>[1]</sup>	74.0330% <sup>[2]</sup>	0.1781% <sup>[3]</sup>

因此，总有 $t_{dec} \leq t_{enc}$ ,  $S_{enc} \leq S_{OG}$ . 这是符合预期和上面算法描述的。

## 结论

本程序能够正常运行并达到实训要求，同时在核心和程序使用上有创新。

## 六、综合实验总结

Huffman编码是一个经典的Entropy-Based的2-Pass压缩算法，关于它的各种语言的实现层出不穷。但Huffman编码一般用于文本的压缩，而且需要预先知道文件内各个符号的统计分布，若需要压缩 各类文件 (i.e. 以二进制方式压缩)，实时 压缩提高程序运行效率，则需要在实现上作出改进。

据此，J. S. Vitter在1987年提出一种改进的Huffman算法，成为Vitter算法。这个算法解决了我们后半部分的问题——实时压缩，提高效率；通过对各类文件进行预处理，以8bits为1byte的方式，我们成功地将各类文件都能作为这个算法的输出，并且获得良好的运行结果。

综上，得益于简单的数据结构和纯C实现的代码，尽管更新树操作相当耗时，在运行时间上和一些传统Huffman实现相比仍更胜一筹；得益于自适应Huffman编码和一些底层处理，程序在压缩率和压缩效率上相比传统Huffman都有一些提升，同时具备了实时编解码的能力，能够处理直播流。

## 功能拓展和改进

- Vitter算法——提高压缩率和压缩效率，具备实时编解码的能力
- 纯C实现——提高程序运行效率
- 命令行参数——取代交互式界面，便于脚本调用，程序化使用；贴合Unix工具程序设计理念。
  - 得益于此，配合类Unix系统自带的 `tar` 一行命令即可对文件夹和多个文件进行压缩。
- 日志功能——将压缩信息存储到文本文件中，便于后续查看具体信息。

## 寻求的改进

- 可以在命令行的基础上增加GUI界面，利用跨平台GUI框架例如Qt完成全平台的构建。
- Vitter算法每次更新树的操作较为耗费时间，可以寻找更优的算法实现如CABAC(Context-based Adaptive Binary Arithmetic Coding)，这正是H.264视频压缩标准（使用上很普遍）使用的算法。

## REFERENCE

1. J. S. Vitter, "Design and Analysis of Dynamic Huffman Codes", Journal of the ACM, 34(4), October 1987, pp 825–845.
2. A. Desoky and M. Gregory, "Compression of text and binary files using adaptive Huffman coding techniques", Conference Proceedings '88., IEEE Southeastcon, Knoxville, TN, USA, 1988, pp. 660-663, doi: 10.1109/SECON.1988.194940

## APPENDIX: SOURCE

源代码以Unix风格写成，函数注释符合Doxygen格式。使用一些POSIX API。

- `huff.h`

```
1  #pragma once
2
3  #include <assert.h>
4  #include <stdarg.h> //va_list, va_start, va_end
5  #include <stdbool.h>
6  #include <stdint.h>
7  #include <stdio.h>
8  #include <stdlib.h>
9  #include <string.h>
10 #include <sys/types.h>
11 #include <unistd.h>
12
13 // return value of the program. 0 = OK, -1 = error
14 #define AHEDOK 0
15 #define AHEDFail -1
16
17 #define timing(a, b) \
18     start = clock(); \
19     a; \
20     diff = clock() - start; \
21     msec = (double)diff * 1e3 / CLOCKS_PER_SEC; \
22     printf("\nTime elapsed: %lfms during %s process\n", msec, b)
23
24 // Encoder/decoder log
25 typedef struct {
26     /* size of the decoded string */
27     uint64_t uncodedSize;
28     /* size of the encoded string */
```

```

29     uint64_t codedSize;
30     /*symbols count*/
31     short n_symbols;
32 } tAHED;
33
34 /**
35  * Encodes the input file, stores the result to the output file and log actions.
36  * @param {tAHED*} ahed encoding log
37  * @param {FILE*} inputFile decoded input file
38  * @param {FILE*} outputFile encoded output file
39  * @return 0 if encoding went OK, -1 otherwise
40  */
41 int AHEDEncoding(tAHED* ahed, FILE* inputFile, FILE* outputFile);
42 /**
43  * Decodes the input file, store result to the output file and log actions.
44  * @param {tAHED*} ahed decoding log
45  * @param {FILE*} inputFile encoded input file
46  * @param {FILE*} outputFile decoded output file
47  * @return 0 decoding was OK, -1 otherwise
48  */
49 int AHEDDecoding(tAHED* ahed, FILE* inputFile, FILE* outputFile);
50
51 /**
52  * Print error to stderr and exit.
53  */
54 void AHEDError(const char* fmt, ...);

```

- huff.c ——程序核心功能实现

```

1  #include "huff.h"
2  // node structure
3  typedef struct node t_node;
4  struct node {
5      int freq;
6      int character;
7      int order;
8      t_node* left;
9      t_node* right;
10     t_node* parent;
11 };
12
13 #define ZERO_NODE -1 // ZERO node indicator
14 #define INNER_NODE -2 // inner node indicator
15 #define MAX_NODES 513 // maximum number of nodes in the tree
16 #define MAX_CODE_LENGTH 513 // maximum number of length of string code in a node
17
18 unsigned char outputBuffer = 0; // byte buffer for output
19 int outputBufferPos = 7; // position in byte buffer; 1byte = 8bit
20

```



```

21 unsigned char inputBuffer = 0; // input buffer
22 int inputBufferPos = -1; // input buffer position
23
24 /**
25  * Get next bit from input stream.
26  * @return bit or EOF
27  * (used only by decoder)
28  */
29 int AHEDGetInputBufferNextBit(FILE* file)
30 {
31     int c;
32     if (inputBufferPos == -1) { // need to get another byte
33         c = getc(file);
34         if (c == EOF)
35             return EOF; // finished reading.
36         inputBuffer = (unsigned char)c;
37         inputBufferPos = 7;
38     }
39
40     int ret = (inputBuffer >> inputBufferPos) & 1;
41     inputBufferPos--;
42
43     return ret;
44 }
45
46 /**
47  * Get byte from input stream or EOF.
48  * @return byte or EOF
49  * (used only by decoder)
50  */
51 int AHEDGetInputBufferChar(FILE* file)
52 {
53     int ret = 0;
54     int i = 7;
55     int next_bit; // next buffer bit
56
57     for (; i ≥ 0; i--) {
58         next_bit = AHEDGetInputBufferNextBit(file);
59         if (next_bit == 0)
60             ret &= (~(1 << i));
61         else if (next_bit == 1)
62             ret |= (1 << i);
63         else
64             return EOF;
65     }
66     return ret;
67 }
68
69 /**

```

```

70  * Put bit (value) to the output buffer.
71  * @return 1 if whole byte was appended, 0 otherwise
72  * (used only by encoder)
73  */
74  int AHEDPutBit2Buffer(FILE* file, int value)
75  {
76      if (value)
77          outputBuffer |= (1 << outputBufferPos);
78      else
79          outputBuffer &= ~(1 << outputBufferPos);
80
81      outputBufferPos--;
82
83      if (outputBufferPos == -1) {
84          putc(outputBuffer, file);
85          outputBufferPos = 7;
86          outputBuffer = 0;
87          return 1;
88      }
89      return 0;
90  }
91
92  /**
93   * Put byte to the output buffer.
94   * @return number of appended bytes (i.e. always 1).
95   * (used only by encoder)
96   */
97  int AHEDPutChar2Buffer(FILE* file, unsigned char c)
98  {
99      int i = 7;
100     for (; i ≥ 0; i--)
101         AHEDPutBit2Buffer(file, (c >> i) & 1);
102     return 1;
103 }
104
105 /**
106  * @return true if c is in the tree, false otherwise
107  * (used only by encoder)
108  */
109  int AHEDFirstInput(t_node** ta, int c)
110  {
111      int i;
112      for (i = 0; i < MAX_NODES; i++) {
113          if (ta[i] == NULL) // reached end of the array
114              return 1;
115          if (ta[i]→character == c)
116              return 0;
117      };
118      return 1;

```

```

119 }
120
121 /**
122  * Append node code n to the output buffer.
123  * @return number of appended bytes
124  * (used only by encoder)
125  */
126 int AHEDOutputNodeCode(FILE* file, t_node* n)
127 {
128     int writeBytes = 0;
129
130     int i = 0;
131     char code[MAX_CODE_LENGTH];
132
133     t_node* tmp = n;
134     while (tmp->parent != NULL) {
135         if (tmp->parent->left == tmp)
136             code[i++] = '0';
137         else if (tmp->parent->right == tmp)
138             code[i++] = '1';
139         tmp = tmp->parent;
140     }
141
142     // output of the code is in reverse order
143     while (--i ≥ 0)
144         writeBytes += AHEDPutBit2Buffer(file, code[i] - '0');
145
146     return writeBytes;
147 }
148
149 /**
150  * Append code of character c to the output buffer.
151  * @return number of appended bytes (i.e. 1)
152  * (used only by encoder)
153  */
154 int AHEDOutputCharCode(FILE* file, unsigned char c)
155 {
156     return AHEDPutChar2Buffer(file, c);
157 }
158
159 /**
160  * Update tree.
161  * Increase frequency of parents or update Huffman tree if needed.
162  * @param {tree_array} array of nodes sorted by `node->order` increasingly.
163  */
164 void AHEDActualizeTree(t_node** tree_array, t_node* actual_node)
165 {
166     t_node* node = actual_node;
167     t_node* sfho_node = NULL; // node with the same frequency and higher order

```

```

168     t_node* aux; // auxiliary pointer for nodes swap
169     t_node* auxpar;
170     t_node* node_parent_left;
171     t_node* node_parent_right;
172
173     int auxorder; // auxiliary variable for order swap
174     int i;
175
176     // till node is not the root
177     while (node->parent != NULL) {
178
179         // find the last node with the same frequency and higher order
180         // (skip parent)
181         sfho_node = NULL;
182         i = node->order - 1;
183         for (; i ≥ 0; i--) {
184             if (tree_array[i]->freq == node->freq
185                 && tree_array[i] != node->parent)
186                 sfho_node = tree_array[i];
187         }
188
189         if (sfho_node != NULL) {
190             // swap node and sfho_node
191
192             // swap subtrees
193             assert(node != sfho_node);
194
195             node_parent_left = node->parent->left;
196             node_parent_right = node->parent->right;
197
198             if (sfho_node->parent->left == sfho_node)
199                 sfho_node->parent->left = node;
200             else if (sfho_node->parent->right == sfho_node)
201                 sfho_node->parent->right = node;
202
203             if (node_parent_left == node) {
204                 node->parent->left = sfho_node;
205             } else if (node_parent_right == node)
206                 node->parent->right = sfho_node;
207
208             // swap parent pointers
209             auxpar = node->parent;
210             node->parent = sfho_node->parent;
211             sfho_node->parent = auxpar;
212
213             // update tree_array
214             tree_array[sfho_node->order] = node;
215             tree_array[node->order] = sfho_node;
216

```

```

217         // update order
218         auxorder = node->order;
219         node->order = sfho_node->order;
220         sfho_node->order = auxorder;
221     }
222
223     // higher frequency
224     node->freq++;
225
226     // go one level up
227     node = node->parent;
228 }
229
230 // increase frequency in the root node
231 node->freq++;
232 }
233
234 /**
235  * Encode the input file, save the result to the output file and log actions.
236  * @param {tAHED} ahed encoding log
237  * @param {FILE*} inputFile decoded input file
238  * @param {FILE*} outputFile encoded output file
239  * @return 0 if encoding went OK, -1 otherwise
240  */
241 int AHEDEncoding(tAHED* ahed, FILE* inputFile, FILE* outputFile)
242 {
243     if (ahed == NULL) {
244         AHEDError("record of coder / decoder was not allocated");
245         return AHEDFail;
246     }
247     ahed->uncodedSize = 0;
248     ahed->codedSize = 0;
249     ahed->n_symbols = 0;
250
251     int i;
252     t_node* tree_array[MAX_NODES]; // nodes array, for easier order recognition and tree
walking
253     for (i = 0; i < MAX_NODES; i++)
254         tree_array[i] = NULL;
255
256     t_node* tree_root = NULL;
257
258     // create ZERO node
259     t_node* zero_node = malloc(sizeof(t_node));
260     if (zero_node == NULL) {
261         AHEDError("not enough memory");
262         return AHEDFail;
263     }
264     zero_node->freq = 0;

```

```

265 zero_node→character = ZERO_NODE;
266 zero_node→order = 0;
267 zero_node→left = NULL;
268 zero_node→right = NULL;
269 zero_node→parent = NULL;
270
271 tree_root = zero_node; // ZERO is the root
272 tree_array[0] = zero_node;
273
274 int c; // read symbol
275 while ((c = getc(inputFile)) ≠ EOF) {
276     ahead→uncodedSize++;
277
278     if (AHEDFirstInput(tree_array, c)) {
279         // Character c was seen for the first time.
280
281         ahead→n_symbols++;
282         ahead→codedSize += AHEDOutputNodeCode(outputFile, zero_node);
283         ahead→codedSize += AHEDOutputCharCode(outputFile, (unsigned char)c);
284
285         // update ZERO → move it down the tree
286         zero_node→order = zero_node→order + 2;
287
288         // create a new node with the character c
289         t_node* nodeX = malloc(sizeof(t_node));
290         if (nodeX == NULL) {
291             AHEDError("not enough memory");
292             return AHEDFail;
293         }
294         nodeX→freq = 1;
295         nodeX→character = c;
296         nodeX→order = zero_node→order - 1;
297         nodeX→left = NULL;
298         nodeX→right = NULL;
299
300         // create new inner node
301         t_node* nodeU = malloc(sizeof(t_node));
302         if (nodeU == NULL) {
303             AHEDError("not enough memory");
304             return AHEDFail;
305         }
306         nodeU→freq = 0;
307         nodeU→character = INNER_NODE;
308         nodeU→order = zero_node→order - 2; // will replace ZERO
309         nodeU→left = zero_node;
310         nodeU→right = nodeX;
311         nodeU→parent = zero_node→parent; // parent node of previous ZERO
312
313         // connect the new node to the left child of ZERO parent

```

```

314         // ZERO goes left down in the tree
315         if (zero_node->parent != NULL)
316             zero_node->parent->left = nodeU;
317
318         nodeX->parent = nodeU;
319         zero_node->parent = nodeU;
320
321         // update tree array
322         tree_array[nodeU->order] = nodeU;
323         tree_array[nodeX->order] = nodeX;
324         tree_array[zero_node->order] = zero_node;
325
326         AHEDActualizeTree(tree_array, nodeU);
327
328     } else {
329         // Character c is already in the tree.
330
331         t_node* nodeX = NULL;
332         // find node with the character c
333         for (i = 0; i < MAX_NODES; i++) {
334             assert(tree_array[i] != NULL);
335             if (tree_array[i]->character == c) {
336                 nodeX = tree_array[i];
337                 break;
338             }
339         }
340         assert(nodeX != NULL);
341         assert(nodeX->parent != NULL); // it cannot be the root node
342         ahead->codedSize += AHEDOutputNodeCode(outputFile, nodeX);
343         AHEDActualizeTree(tree_array, nodeX);
344     }
345 }
346
347 // align to the byte with zeros
348 while (outputBufferPos != 7) {
349     ahead->codedSize += AHEDPutBit2Buffer(outputFile, 0);
350 }
351
352 // free memory occupied by the tree
353 i = 0;
354 for (; i < MAX_NODES; i++) {
355     if (tree_array[i] == NULL)
356         break;
357     free(tree_array[i]);
358 }
359
360 return AHEDOK;
361 }
362

```

```

363  /**
364   * Decodes the input file, stores result to the output file and log actions.
365   * @param {tAHED*} ahed decoding log
366   * @param {FILE*} inputFile encoded input file
367   * @param {FILE*} outputFile decoded output file
368   * @return 0 decoding was OK, -1 otherwise
369   */
370  int AHEDDecoding(tAHED* ahed, FILE* inputFile, FILE* outputFile)
371  {
372      if (ahed == NULL) {
373          AHEDError("record of coder / decoder was not allocated");
374          return AHEDFail;
375      }
376      ahed->uncodedSize = 0;
377      ahed->codedSize = 0;
378      ahed->n_symbols = 0;
379
380      int i;
381      t_node* tree_array[MAX_NODES];
382      for (i = 0; i < MAX_NODES; i++)
383          tree_array[i] = NULL;
384
385      t_node* tree_root = NULL;
386
387      // create ZERO node
388      t_node* zero_node = malloc(sizeof(t_node));
389      if (zero_node == NULL) {
390          AHEDError("not enough memory");
391          return AHEDFail;
392      }
393      zero_node->freq = 0;
394      zero_node->character = ZERO_NODE;
395      zero_node->order = 0;
396      zero_node->left = NULL;
397      zero_node->right = NULL;
398      zero_node->parent = NULL;
399
400      tree_root = zero_node; // ZERO is the root
401      tree_array[0] = zero_node;
402
403      int not_enc_symbol = 1; // is symbol uncompressed?
404
405      int c; // read symbol
406      int end = 0; // decoder end indicator
407      while (!end) {
408
409          if (not_enc_symbol) {
410              // Symbol is uncompressed

```



```

412     ahead->n_symbols++;
413     c = AHEDGetInputBufferChar(inputFile);
414     if (c == EOF)
415         break;
416     ahead->codedSize++;
417
418     putc(c, outputFile); // output one byte
419     ahead->uncodedSize++;
420
421     // update ZERO node order -> move down the tree
422     zero_node->order = zero_node->order + 2;
423
424     // create a new node with character c
425     t_node* nodeX = malloc(sizeof(t_node));
426     if (nodeX == NULL) {
427         AHEDError("not enough memory");
428         return AHEDFail;
429     }
430     nodeX->freq = 1;
431     nodeX->character = c;
432     nodeX->order = zero_node->order - 1;
433     nodeX->left = NULL;
434     nodeX->right = NULL;
435
436     // create a new inner node
437     t_node* nodeU = malloc(sizeof(t_node));
438     if (nodeU == NULL) {
439         AHEDError("not enough memory");
440         return AHEDFail;
441     }
442     nodeU->freq = 0;
443     nodeU->character = INNER_NODE;
444     nodeU->order = zero_node->order - 2; // replaces ZERO
445     nodeU->left = zero_node;
446     nodeU->right = nodeX;
447     nodeU->parent = zero_node->parent; // parent of the previous ZERO node
448
449     // connect the new inner node to the left child of the ZERO parent
450     // ZERO goes to the left down in the tree
451     if (zero_node->parent != NULL)
452         zero_node->parent->left = nodeU;
453
454     nodeX->parent = nodeU;
455     zero_node->parent = nodeU;
456
457     // update order in the tree array
458     tree_array[nodeU->order] = nodeU;
459     tree_array[nodeX->order] = nodeX;
460     tree_array[zero_node->order] = zero_node;

```

```

461
462     AHEDActualizeTree(tree_array, nodeU);
463
464     not_enc_symbol = 0;
465
466     } else {
467         // compressed symbol
468
469         t_node* p_node = tree_array[0];
470
471         // code has to end in a leaf node
472
473         int nextBit;
474
475         while (p_node->right != NULL && p_node->left != NULL) {
476             nextBit = AHEDGetInputBufferNextBit(inputFile);
477             if (nextBit == 1)
478                 p_node = p_node->right;
479             else if (nextBit == 0)
480                 p_node = p_node->left;
481             else { // end of file
482                 end = 1;
483                 break;
484             }
485
486             // input buffer filled → process the next byte
487             if (inputBufferPos == 6)
488                 ahead->codedSize++;
489         }
490         // end of file?
491         if (!end) {
492             // is the next symbol uncompressed?
493             if (p_node->character == ZERO_NODE) {
494                 not_enc_symbol = 1;
495             } else {
496                 // p_node is a leaf node with the character
497                 putc(p_node->character, outputFile);
498                 ahead->uncodedSize++;
499                 AHEDActualizeTree(tree_array, p_node);
500                 not_enc_symbol = 0;
501             }
502         } // endif (!end)
503     }
504 } // endwhile
505
506 // cleanup
507 i = 0;
508 for (; i < MAX_NODES; i++) {
509     if (tree_array[i] == NULL)

```

```

510         break;
511         free(tree_array[i]);
512     }
513
514     return AHEDOK;
515 }
516 /**
517  * Print error to stderr and exit.
518  */
519 void AHEDError(const char* fmt, ...)
520 {
521     va_list args;
522     fprintf(stderr, "AHED ERROR: ");
523
524     va_start(args, fmt);
525     vfprintf(stderr, fmt, args);
526     va_end(args);
527
528     // exit(AHEDFail);
529 }

```

- `main.c` — 命令行实现, Driver Code.

```

1  #include "huff.h"
2  #include <time.h>
3
4  // AHED stands for Adaptive Huffman Encoding.
5
6  int main(int argc, char** argv)
7  {
8
9      char* ifile = NULL;
10     char* ofile = NULL;
11     char* lfile = NULL;
12     int lFlag = 0;
13     int cFlag = 0;
14     int xFlag = 0;
15     int hFlag = 0;
16     int qFlag = 0;
17     int c;
18
19     FILE* inputFile = NULL;
20     FILE* outputFile = NULL;
21     FILE* logFile = NULL;
22
23     opterr = 0;
24
25     // process program arguments
26     while ((c = getopt(argc, argv, "i:o:l:cxhq")) != -1) {

```

```
27     switch (c) {
28     case 'i':
29         ifile = optarg;
30         break;
31     case 'o':
32         ofile = optarg;
33         break;
34     case 'l':
35         lfile = optarg;
36         lFlag = 1;
37         break;
38     case 'c':
39         cFlag = 1;
40         break;
41     case 'x':
42         xFlag = 1;
43         break;
44     case 'h':
45         hFlag = 1;
46         break;
47     case 'q':
48         qFlag = 1;
49         break;
50     case '?':
51     default:
52         AHEDError("unknown argument");
53         return AHEDFail;
54         break;
55     }
56 }
57
58 // help
59 if (hFlag == 1) {
60     printf("USAGE: main -h | -q | -c | -x [-i input_file] [-o output_file] [-l
log_file] \n");
61     return AHEDOK;
62 }
63
64 // input file
65 if (ifile == NULL)
66     inputFile = stdin;
67 else
68     inputFile = fopen(ifile, "rb");
69 if (inputFile == NULL) {
70     AHEDError("can not find an input file");
71     return AHEDFail;
72 }
73
74 // output file
```

```

75     if (ofile == NULL) {
76         outputFile = stdout;
77     } else
78         outputFile = fopen(ofile, "wb");
79     if (outputFile == NULL) {
80         AHEDError("can not open an output file");
81         return AHEDFail;
82     }
83
84     // log file
85     if (lFlag == 1) {
86         if (lfile == NULL) {
87             AHEDError("can not open a log file");
88             return AHEDFail;
89         } else
90             logFile = fopen(lfile, "a");
91     }
92
93     if (cFlag == 1 && xFlag == 1) {
94         AHEDError("please make a decision, encode or decode?\n");
95         return AHEDFail;
96     }
97
98     // encoding/decoding log structure
99     tAHED* ahead = malloc(sizeof(tAHED));
100    if (ahead == NULL) {
101        AHEDError("not enough memory");
102        return AHEDFail;
103    }
104    clock_t start, diff;
105    double msec;
106    if (cFlag == 1) {
107        if (!qFlag)
108            if (ofile)
109                printf("\033[1;33m=====Encoding=====\\033[0m\\n");
110        if (!qFlag)
111            if (ofile)
112                printf("Encode from %s to %s %s\\n", ifile, (ofile ? ofile : "stdout"),
113                    (lFlag == 1 ? "(Logging)" : ""));
114        timing(AHEDEncoding(ahead, inputFile, outputFile), "ENCODING");
115        if (!qFlag)
116            if (ofile)
117                printf("n_Symbols=%d, Encoded Size=%lld B, Decoded Size=%lld B,
118                    Compressed ratio = %lf%%\\n", ahead->n_symbols, ahead->codedSize, ahead->uncodedSize,
119                    (double)ahead->codedSize * 100 / ahead->uncodedSize);
120        if (!qFlag)
121            if (ofile)
122                printf("\033[1;33m=====Completed=====\\033[0m\\n");
123    } else if (xFlag == 1) {

```

```

121         if (!qFlag)
122             if (ofile)
123                 printf("\033[1;33m=====Decoding=====\033[0m\n");
124         if (!qFlag)
125             if (ofile)
126                 printf("Decode from %s to %s %s\n", ifile, (ofile ? ofile : "stdout"),
127 (lFlag == 1 ? "Logging" : ""));
128         timing(AHEDDecoding(ahed, inputFile, outputFile), "DECODING");
129         if (!qFlag)
130             if (ofile)
131                 printf("n_Symbols=%d, Encoded Size=%lld B, Decoded Size=%lld B,
132 Compressed ratio = %lf%%\n", ahed->n_symbols, ahed->codedSize, ahed->uncodedSize,
133 (double)ahed->codedSize * 100 / ahed->uncodedSize);
134         if (!qFlag)
135             if (ofile)
136                 printf("\033[1;33m=====Completed===== \033[0m\n");
137     }
138     // log
139     if (logFile != NULL) {
140         if (cFlag == 1)
141             fprintf(logFile, "Encode from %s to %s", ifile, ofile);
142         else if (xFlag == 1)
143             fprintf(logFile, "Decode from %s to %s", ifile, ofile);
144         fprintf(logFile, "uncodedSize = %lld\n", ahed->uncodedSize);
145         fprintf(logFile, "codedSize = %lld\n", ahed->codedSize);
146         fprintf(logFile, "Compressed ratio = %lf\n", (double)ahed->codedSize / ahed-
147 >uncodedSize);
148         fclose(logFile);
149     }
150     // cleanup
151     free(ahed);
152     if (inputFile != NULL)
153         fclose(inputFile);
154     if (outputFile != NULL)
155         fclose(outputFile);
156
157     return AHEDOK;
158 }

```

- 
1. 之所以不采用 `png` (lossless) 或 `jpeg` (lossy) 是因为它们都有不同程度的压缩，再次压缩效果不明显。位图则是不压缩的原文件，效果明显 ↩
  2. 可执行文件通常不压缩，且空白段多，因此压缩效果 十分 明显 ↩
  3. Word文件本身就是一个压缩包，故而压缩效果 十分 不明显 ↩