Thanks to Owen Astrachan (Duke) and Julie Zelenski for creating this. Modifications by Keith Schwarz, Stuart Reges, Marty Stepp.

感谢 Owen Astrachan（杜克大学）和 Julie Zelenski 制作此文档。修改：Keith Schwarz、Stuart Reges、Marty Stepp。

This assignment focuses on binary trees and priority queues. We provide you with several other support files, but you should not modify them. For example, we provide you with a huffmanmain. cpp that contains the program's overall text menu system; you must implement the functions it calls to perform various file compression / decompression operations.

本作业的重点是二叉树和优先队列。我们为您提供了其他几个支持文件，但您不应修改它们。例如，我们提供的 huffmanmain.cpp 包含程序的整个文本菜单系统；您必须实现它调用的函数，以执行各种文件压缩/解压缩操作。

Due: Monday, November 11^{th} , 2019 at 11:59pm

截止时间：11 月 11^{th}，2019 日星期一 11:59pm

Huffman Encoding 赫夫曼编码

Huffman encoding is an algorithm devised by David A. Huffman of MIT in 1952 for compressing text data to make a file occupy a smaller number of bytes. This relatively simple compression algorithm is powerful enough that variations of it are still used today in computer networks, HDTV, and other applications. Normally text data is stored in a standard format of 8 bits per character using an encoding called ASCII that maps every character to a binary integer value from 0-255.

哈夫曼编码是麻省理工学院的戴维·A·哈夫曼（David A. Huffman）于 1952 年设计的一种算法，用于压缩文本数据，使文件占用的字节数更少。这种相对简单的压缩算法功能强大，其变体至今仍用于计算机网络、高清电视和其他应用中。通常，文本数据以每个字符 8 位的标准格式存储，使用一种称为 ASCII 的编码，该编码将每个字符映射为 0-255 之间的二进制整数值。

The idea of Huffman encoding is to abandon the rigid 8-bits-per-character requirement and use variable-length binary encodings for different characters. The advantage of doing this is that if a character occurs frequently in the file, such as the common letter ' **e** ', it could be given a

哈夫曼编码的理念是放弃每个字符 8 位的硬性规定，对不同字符使用长度可变的二进制编码。这样做的好处是，如果一个字符在文件中频繁出现，例如常见的字母" **e** "，就可以给它一个

```
Welcome to CS 106X Shrink-It!
...
1) build character frequency table
2) build encoding tree
3) build encoding map
4) encode data
5) decode data
C) compress file
D) decompress file
F) free tree memory
B) binary file viewer
T) text file viewer
S) side-by-side file comparison
Q) quit
Your choice? c
Input file name: large.txt
Output file name (Enter for large.huf):
Reading 9768 uncompressed bytes.
Compressing ...
Wrote 5921 compressed bytes.
```

example output from provided HuffmanMain client

所提供的 HuffmanMain 客户端的输出示例

shorter encoding (fewer bits), making the file smaller. The tradeoff is that some characters may need to use encodings that are longer than 8 bits, but this is reserved for characters that occur so infrequently that the extra cost is worth it.

更短的编码（更少的比特），使文件更小。这样做的代价是，有些字符可能需要使用比 8 位更长的编码，但这只适用于不常出现的字符，额外的成本是值得的。

The table below compares ASCII values of various characters to possible Huffman encodings for some English text. Frequent characters such as space and ' **e** ' have short encodings, while rare ones like ' **x** ' and ' **z** ' have longer ones.

下表比较了一些英文文本中各种字符的 ASCII 值和可能的哈夫曼编码。空格和" **e** "等常见字符的编码较短，而" **x** "和" **z** "等罕见字符的编码较长。

Character 人物	ASCII value ASCII 值	ASCII (binary) ASCII（二进制）	Huffman (binary) 赫夫曼（二进制）
' ' ' '	32	00100000	10
'a'	97	01100001	0001
'b'	98	01100010	0111010
'c'	99	01100011	001100
'e'	101	01100101	1100
'z' 'z'	122	01111010	00100011010

The steps involved in Huffman encoding a given text source file into a destination compressed file are:

将给定的文本源文件转换为目标压缩文件的哈夫曼编码步骤如下：

count frequencies: Examine a source file's contents and count the number of occurrences of each character.

计算频率检查源文件的内容并计算每个字符出现的次数。

build encoding tree: Build a binary tree with a particular structure, where each leaf node stores a character and its frequency count. A priority queue is used to help build the tree along the way.

构建编码树：构建具有特定结构的二叉树，其中每个叶节点存储一个字符及其频率计数。在此过程中，优先级队列会帮助建立编码树。

build encoding map: Traverse the binary tree to discover the binary encodings of each character.

构建编码图：遍历二叉树，找出每个字符的二进制编码。

encode data: Re-examine the source file's contents, and for each character, output the encoded binary version of that character to the destination file.

编码数据：重新检查源文件的内容，并为每个字符向目标文件输出该字符的二进制编码版本。

Encoding a File, Step 1: Counting Frequencies

为文件编码，第 1 步：计算频率

For example, suppose we have a file example.txt whose contents are: **ababcab** In the original file, this text occupies 10 bytes, or 80 bits, of data. The 10^{th} is a special "end-of-file" (EOF) byte.

例如，假设我们有一个文件 example.txt，其内容为 **ababcab** 在原始文件中，这段文字占 10 个字节，即 80 位数据。 10^{th} 是一个特殊的 "文件结束"（EOF）字节。

byte 字节	1	2	3	4	5	6	7	8	9	10
char 烧焦	'a'	'b'	, '	'a'	'b'	,	'c'	'a'	'b'	EOF
ASCII	97	98	32	97	98	32	99	97	98	256
binary 二进制	01100001	01100010	00100000	01100001	01100010	00100000	01100011	01100001	01100010	N/A 不适用

In Step 1 of Huffman's algorithm, a count for each character is computed. The counts are represented as a map:

在哈夫曼算法的第 1 步，计算每个字符的计数。计数以映射的形式表示：

{' ':2, 'a':3, 'b':3, 'c': 1, EOF: 1}

Encoding a File, Step 2: Building an Encoding Tree

编码文件，第 2 步：构建编码树

Step 2 of Huffman's algorithm places our counts into binary tree nodes, with each node storing a character and a count of its occurrences. The nodes are then put into a priority queue, which keeps them in prioritized order with smaller counts having higher priority, so that characters with lower counts will come out of the queue sooner. The priority queue is

哈夫曼算法的第 2 步将计数放入二叉树节点，每个节点存储一个字符及其出现次数。然后将这些节点放入优先级队列，按照优先级顺序排列，计数越小优先级越高，这样计数较低的字符会更快从队列中出来。优先队列为

somewhat arbitrary in how it breaks ties, such as ' **c** ' being before EOF and ' **a** ' being before 'b'.

在如何打破并列关系方面有些武断，例如" **c** "位于 EOF 之前，而" **a** "位于 "b "之前。

```
  front                                 back
+-----+    +-----+    +-----+    +-----+    +-----+
| 'c' |    | EOF |    | ' ' |    | 'a' |    | 'b' |
|  1  |    |  1  |    |  2  |    |  3  |    |  3  |
+-----+    +-----+    +-----+    +-----+    +-----+
```

Now the algorithm repeatedly removes the two nodes from the front of the queue-the two with the smallest frequencies-and marries them into a new node whose frequency is their sum. The two nodes are wired in as children of the new node; the first removed becomes the left child, and the second becomes the right. The new node is re-inserted into the queue in sorted order. This process is repeated until the queue contains only one binary tree node with all the others as its children. This will be the root of our final Huffman tree. The following diagram shows this process:

现在，算法会反复从队列前方移除两个节点--频率最小的两个节点--并将它们连接到一个新节点中，新节点的频率就是这两个节点的频率之和。这两个节点作为新节点的子节点被连接进来；第一个被移除的节点成为左节点，第二个节点成为右节点。新节点按排序顺序重新插入队列。这个过程不断重复，直到队列中只有一个二叉树节点，其他节点都是它的子节点。这就是我们最终的哈夫曼树的根节点。下图展示了这一过程：

1) 'c' node and EOF node are removed and joined

```
+-----+   +-----+   +-----+   +-----+
| ' ' |   |     |   | 'a' |   | 'b' |
|  2  |   |  2  |   |  3  |   |  3  |
+-----+   +-----+   +-----+   +-----+
            / \
           /   \
     +-----+   +-----+
     | 'c' |   | EOF |
     |  1  |   |  1  |
     +-----+   +-----+
```

2) ' ' node and c/EOF node are removed and joined

```
+-----+   +-----+   +-----+
| 'a' |   | 'b' |   |     |
|  3  |   |  3  |   |  4  |
+-----+   +-----+   +-----+
                      / \
                     /   \
               +-----+   +-----+
               | ' ' |   |     |
               |  2  |   |  2  |
               +-----+   +-----+
                            / \
                           /   \
                     +-----+   +-----+
                     | 'c' |   | EOF |
                     |  1  |   |  1  |
                     +-----+   +-----+
```

3) 'a' and 'b' nodes are removed and joined

```
             +-----+                +-----+
             |     |                |     |
             |  4  |                |  6  |
             +-----+                +-----+
               / \                    / \
              /   \                  /   \
        +-----+   +-----+      +-----+   +-----+
        | ' ' |   |     |      | 'a' |   | 'b' |
        |  2  |   |  2  |      |  3  |   |  3  |
        +-----+   +-----+      +-----+   +-----+
                    / \
                   /   \
             +-----+   +-----+
             | 'c' |   | EOF |
             |  1  |   |  1  |
             +-----+   +-----+
```

4) ' '/c/EOF node and a/b node are removed/joined

```
                 +-----+
                 |     |
                 | 10  |
                 +-----+
                /       \
               /         \
        +-----+           +-----+
        |     |           |     |
        |  4  |           |  6  |
        +-----+           +-----+
          / \               / \
         /   \             /   \
  +-----+   +-----+   +-----+   +-----+
  | ' ' |   |     |   | 'a' |   | 'b' |
  |  2  |   |  2  |   |  3  |   |  3  |
  +-----+   +-----+   +-----+   +-----+
              / \
             /   \
       +-----+   +-----+
       | 'c' |   | EOF |
       |  1  |   |  1  |
       +-----+   +-----+
```

Encoding a File, Step 3: Building an Encoding Map

编码文件，第 3 步：构建编码图

The Huffman code for each character is derived from your binary tree by thinking of each left branch as a bit value of 0 and each right branch as a bit value of 1 , as shown in the diagram at right. The code for each character can be determined by traversing the tree. To reach ' ', we go left twice from the root, so the code for ' ' is **00**. The code for ' **c** ' is **010**, the code for **EOF** is **011**, the code for ' **a** ' is **10** and the code for ' **b** ' is **11**. By traversing the tree, we can produce a map from characters to their binary representations.

如右图所示，每个字符的哈夫曼代码都是从二进制树中导出的，方法是将每个左分支视为位值 0，而将每个右分支视为位值 1。每个字符的代码可以通过遍历树来确定。要到达""，我们要从根部向左走两次，因此""的代码为 **00** 。 **c** ' 的代码是 **010** ， **EOF** 的代码是 **011**， **a** ' 的代码是 **10** ， **b** ' 的代码是 **11** 。通过遍历这棵树，我们可以生成从字符到其二进制表示的映射。

```
                +-----+
                |     |
                | 10  |
                +-----+
               /       \
         0   /           \   1
      +-----+             +-----+
      |     |             |     |
      |  4  |             |  6  |
      +-----+             +-----+
         / \                 / \
  0   /   \   1       0   /   \   1
+-----+   +-----+   +-----+   +-----+
| ' ' |   |     |   | 'a' |   | 'b' |
|  2  |   |  2  |   |  3  |   |  3  |
+-----+   +-----+   +-----+   +-----+
             / \
       0   /   \   1
      +-----+   +-----+
      | 'c' |   | EOF |
      |  1  |   |  1  |
      +-----+   +-----+
```

Though the binary representations are integers, since they consist of binary digits and can be arbitrary length, we will store them as strings. For this tree, it would be:

虽然二进制表示法是整数，但由于它们由二进制数字组成，长度也可以是任意的，因此我们将它们存储为字符串。对于这棵树来说，它应该是

{' ':"00", 'a':"10", 'b':"11", 'c':"010", EOF: "011"}

Encoding a File, Step 4: Encoding the Text Data:

编码文件，第 4 步：编码文本数据：

Using the encoding map, we can encode the file's text into a shorter binary representation. Using the preceding encoding map, the text **ababcab** would be encoded as:

使用编码映射，我们可以将文件文本编码为更短的二进制表示法。使用前面的编码映射，文本 **ababcab** 将被编码为

1011001011000101011011

The following table details the char-to-binary mapping in more detail. The overall encoded contents of the file require 22 bits, or almost 3 bytes, compared to the original file of 10 bytes.

下表详细介绍了字符到二进制的映射。与原始文件的 10 字节相比，文件的总体编码内容需要 22 位，即近 3 字节。

char 烧焦	'a'	'b'	' '	'a'	'b'	' '	'c'	'a'	'b'	EOF
binary 二进制	10	11	00	10	11	00	010	10	11	011

Since the character encodings have different lengths, often the length of a Huffmanencoded file does not come out to an exact multiple of 8 bits. Files are stored as sequences of whole bytes, so in cases like this the remaining digits of the last byte are filled with zeroes. You do not need to worry about this; it is managed for you as part of the underlying file system.

由于字符编码的长度不同，哈夫曼编码文件的长度往往不是 8 位的精确倍数。文件是以整字节序列的形式存储的，因此在这种情况下，最后一个字节的剩余数字会被填为 0。你无需担心这个问题，底层文件系统会为你管理这个问题。

byte	1	2	3
char	a b a	b c a	b EOF
binary	10 11 00 10	11 00 010 1	0 11 011 00

It might worry you that the characters are stored without any delimiters between them, since their encodings can be different lengths and characters can cross byte boundaries, as with ' **a** ' at the end of the second byte. But this will not cause problems in decoding the file, because Huffman encodings by definition have a useful prefix property where no character's encoding can be the prefix of another's.

由于字符的编码长度可能不同，而且字符可能跨越字节边界，例如第二个字节末尾的" **a** "，因此在存储字符时它们之间没有任何分隔符，这可能会让您担心。但这不会给文件解码带来问题，因为根据定义，哈夫曼编码有一个有用的前缀属性，即任何字符的编码都不能是另一个字符的前缀。

Decoding a File 解码文件

You can use a Huffman tree to decode text that was previously encoded with its binary patterns. The decoding algorithm is to read each bit from the file, one at a time, and use this bit to traverse the Huffman tree. If the bit is a 0 , you move left in the tree. If the bit is 1 , you move right. You do this until you hit a leaf node. Leaf nodes represent characters, so once you reach a leaf, you output that character. For example, suppose we are given the same encoding tree above, and we are asked to decode a second file containing the following bits:

您可以使用哈夫曼树来解码之前用二进制模式编码的文本。解码算法是从文件中读取每一位，每次读取一位，并使用该位遍历哈夫曼树。如果该位为 0，则在树中向左移动。如果位为 1，则向右移动。这样做直到遇到叶节点为止。叶节点代表字符，因此一旦到达叶节点，就会输出该字符。例如，假设我们得到了上述相同的编码树，要求我们解码包含以下位的第二个文件：

1110010001001010011

Using the Huffman tree, we walk from the root until we find characters, then output them and go back to the root.

使用哈夫曼树，我们从树根开始走，直到找到字符，然后输出字符并返回树根。

We read a 1 (right), then a 1 (right). We reach ' **b** ' and output b. Back to root.

我们读取一个 1（右），然后读取一个 1（右）。读到" **b** "，输出 b。

1110010001001010011

We read a 1 (right), then a 0 (left). We reach 'a' and output a. (Back to root.)

我们读取一个 1（右），然后读取一个 0（左）。读到 "a"，输出 a。

1110010001001010011

We read a 0 (left), then a 1 (right), then a 0 (left). We reach ' **c** ' and output c.

我们读取 0（左），然后是 1（右），接着是 0（左）。读到" **c** "，输出 c。

1110010001001010011

We read a 0 (left), then a 0 (left). We reach ' ' and output a space.

我们读取一个 0（左），然后读取一个 0（左）。我们读到""，然后输出一个空格。

1110010001001010011

We read a 1 (right), then a 0 (left). We reach ' a' and output a.

我们读取一个 1（右侧），然后读取一个 0（左侧）。读到 "a"，输出 a。

1110010001001010011

We read a 0 (left), then a 1 (right), then a 0 (left). We reach ' **c** ' and output **c**.

我们读取 0（左），然后是 1（右），接着是 0（左）。我们读到" **c** "，然后输出 **c** 。

1110010001001010011

We read a 1 (right), then a 0 (left). We reach ' a' and output a.

我们读取一个 1（右侧），然后读取一个 0（左侧）。读到 "a"，输出 a。

1110010001001010011

We read a $0, 1, 1$. This is our EOF encoding pattern, so we stop. The overall decoded text is bac aca.

我们读取一个 $0, 1, 1$ 。这是我们的 EOF 编码模式，因此我们停止。整个解码文本为 bac aca。

Provided Code 提供代码

We provide you with a file HuffmanNode. **h** that declares some useful support code, including the HuffmanNode structure, which represents a node in a Huffman encoding tree.

我们为您提供了一个文件 HuffmanNode。 **h**，该文件声明了一些有用的支持代码，包括 HuffmanNode 结构，该结构表示哈夫曼编码树中的一个节点。

```
struct HuffmanNode {
    int character; // character being represented by this node
    int count; // number of occurrences of that character
    HuffmanNode* zero; // 0 (left) subtree (NULL if empty)
    HuffmanNode* one; // 1 (right) subtree (NULL if empty)
    ...
};
```

The character field is declared as type int, but you should think of it as a char. (Types char and int are largely interchangeable in C++, but using int here allows us to

字符字段声明为 int 类型，但您应该将其视为 char。(在 C++ 中，char 和 int 类型在很大程度上是可以互换的，但在这里使用 int 可以让我们

sometimes use character to store values outside the normal range of char, for use as special flags.) The character field can take one of three types of values:

有时使用字符来存储 char 正常范围之外的值，以用作特殊标记）。字符字段可存储三种值之一：

an actual char value; 一个实际字符值；
the constant PSEUDO_EOF (defined in bitstream.h in the Stanford library), which represents the pseudo-EOF value that you will need to place at the end of an encoded stream; or

常量 PSEUDO_EOF（定义在 Stanford 库的 bitstream.h），它表示需要放在编码流末尾的伪EOF 值；或

the constant NOT_A_CHAR (defined in bitstream. **h** in the Stanford library), which represents something that isn't actually a character. This can be stored in interior nodes of the Huffman encoding tree, because such nodes do not represent any one individual character.

常量 NOT_A_CHAR（定义于斯坦福库中的 bitstream.这可以存储在哈夫曼编码树的内部节点中，因为这些节点并不代表任何一个单独的字符。

Bit Input/Output Streams 比特输入/输出流

In parts of this program you will need to read and write bits to files. In the past we have wanted to read input an entire line or word at a time, but for this program, it's much better to read one single character (byte) at a time. You should use the following in/output stream functions:

在本程序的部分内容中，你需要读取和写入文件的位。过去，我们希望一次读取整行或整字的输入，但在本程序中，一次读取一个字符（字节）会更好。您应该使用以下输入/输出流函数：

ostream (output stream) member 输出流	Description 说明
void put(int byte)	writes a single byte (character, 8 bits) to the output stream 将单字节（字符，8 位）写入输出流

istream (input stream) member istream（输入流）成员	Description 说明
int get( )	reads a single byte (character, 8 bits) from input (or EOF) 从输入（或 EOF）读取单字节（字符，8 位）

You might also find that you want to read an input stream, then "rewind" it back to the start and read it again. To do this on an input stream variable named input, you can use the rewindStream function from filelib.h:

您可能还会发现，您想读取一个输入流，然后将其 "倒带 "回起点并再次读取。要在名为 input 的输入流变量上实现这一功能，可以使用 filelib.h 中的 rewindStream 函数：

```
rewindStream(input); // tells the stream to seek back to the beginning
```

To read or write a compressed file, even a whole byte is too much! You will want to read and write binary data one single bit at a time, which is not directly supported by the default in/output streams. Fortunately, the Stanford C++ library provides obitstream and ibitstream classes with writeBit and readBit members to make it easier.

要读取或写入压缩文件，即使是整个字节也太多了！您需要一次读写一个比特的二进制数据，而默认的输入/输出流并不直接支持这一点。幸运的是，斯坦福 C++ 程序库提供了带有 writeBit 和 readBit 成员的 obitstream 和 ibitstream 类，让这一切变得更容易。

obitstream (bit output stream) member 成员	Description 说明
void writeBit(int bit)	writes a single bit (0 or 1) to the output stream 将一个比特（0 或 1）写入输出流

ibitstream (bit input stream) member 比特流（比特输入流）成员	Description 说明
int readBit()	reads a single bit (0 or 1) from input; −1 at end of file 从输入端读取单个位（0 或 1）； −1 在文件末尾

When reading from an bit input stream (ibitstream), you can detect the end of the file by either looking for a readBit result of -1 , or by calling the **fail** member function on the input stream after trying to read from it, which will return true if the last readBit call was unsuccessful because the end of the file had previously been reached.

从比特输入流 (ibitstream)读取数据时，可以通过查找 readBit 结果为 -1，或者在尝试读取数据后调用输入流上的 **fail** 成员函数来检测文件是否已结束。

Note that the bit in/output streams also provide the same members as the original ostream and istream classes from the C++ standard library, such as getline, <<, >>, etc. But you won't want to use them, because they operate on an entire byte at a time, or more, whereas you want to process these streams one bit at a time.

请注意，比特输入/输出流也提供了与 C++ 标准库中原始 ostream 和 istream 类相同的成员，如 getline、<<、>> 等。但你不会想使用它们，因为它们一次对整个字节或更多字节进行操作，而你想一次处理这些流中的一个比特。

Implementation Details 实施细节

In this assignment you will write the following functions in the file encoding. cpp to encode and decode data using the Huffman algorithm described previously. Our provided main client program will allow you to test each function one at a time before moving on to the next. You must write the following functions, although you can add more functions as helpers if you like. You will find yourself traversing a binary tree from top to bottom for a few of these methods. You can do the traversal recursively or iteratively, whichever approach seems cleaner to you.

在本作业中，您将在 encoding.cpp 文件中编写以下函数，使用前面描述的哈夫曼算法对数据进行编码和解码。我们提供的主客户端程序将允许您一次测试一个函数，然后再测试下一个函数。您必须编写以下函数，当然您也可以添加更多的函数作为辅助函数。您会发现自己需要为其中的几个方法从上到下遍历一棵二叉树。你可以采用递归或迭代的方式进行遍历，无论哪种方式对你来说更简洁。

Map<int, int> buildFrequencyTable(istream& input)
This is Step 1 of the encoding process. In this function you read input from a given istream (which could be a file on disk, a string buffer, etc.). You should count and return a mapping from each character (represented as int here) to the number of times that character appears in the file. You should also add a single occurrence of the fake character PSEUDO_EOF into your map. You may assume that the input file exists and can be read, though the file might be empty. An empty file would cause you to return a map containing only the 1 occurrence of PSEUDO_EOF.

这是编码过程的第一步。在此函数中，您从给定的 istream（可以是磁盘上的文件、字符串缓冲区等）中读取输入。您应计算并返回每个字符（此处用 int 表示）与该字符在文件中出现次数的映射。您还应在映射中添加一个假字符 PSEUDO_EOF。您可以假设输入文件存在并且可以读取，尽管文件可能是空的。如果文件为空，您将返回一个只包含 1 次 PSEUDO_EOF 的映射。

HuffmanNode *buildEncodingTree(const Map<int, int>& freqTable)
This is Step 2 of the encoding process. In this function you will accept a frequency table (like the one you built in buildFrequencyTable) and use it to create a Huffman encoding tree based on those frequencies. Return a pointer to the node representing the root of the tree.

这是编码过程的第二步。在此函数中，您将接受一个频率表（如在 buildFrequencyTable 中创建的频率表），并使用它根据这些频率创建一个哈夫曼编码树。返回一个指向代表编码树根节点的指针。

You may assume that the frequency table is valid: that it does not contain any keys other than char values and PSEUDO_EOF. All counts are positive integers; it contains at least one key/value pairing; etc.

您可以假定频率表是有效的：它不包含除字符值和 PSEUDO_EOF 以外的任何键。所有计数都是正整数；至少包含一个键/值配对；等等。

When building the encoding tree, you will need to use a priority queue to keep track of which nodes to process next. Use the PriorityQueue collection provided by the Stanford libraries, defined in library header pqueue. **h**. This priority queue allows each element to be enqueued along with an associated priority. The priority queue then sorts elements by their priority, with the dequeue function always returning the element with the minimum priority number. Consult the docs on the course website and lecture slides for more information about priority queues. Note that this is not exactly the pqueue. **h** file you worked with for Assignment 5. This is the one that comes with the Stanford libraries.

在构建编码树时，需要使用优先级队列来跟踪下一步处理哪个节点。请使用斯坦福库提供的 PriorityQueue 集合，该集合在库头 pqueue 中定义。 **h** 中定义。该优先级队列允许每个元素与相关的优先级一起排队。然后，优先级队列按优先级对元素进行排序，而 dequeue 函数总是返回优先级最小的元素。有关优先级队列的更多信息，请查阅课程网站上的文档和讲座幻灯片。请注意，这并不完全是 pqueue。 **h** 文件。这是 Stanford 库自带的文件。

```
Map<int, string> buildEncodingMap(HuffmanNode *encodingTree)
```

This is Step 3 of the encoding process. In this function you will accept a pointer to the root node of a Huffman tree (like the one you built in buildEncodingTree) and use it to create and return a Huffman encoding map based on the tree's structure. Each key in the map is a character, and each value is the binary encoding for that character represented as a string. For example, if the character ' **a** ' has binary value **10** and ' **b** ' has **11**, the map should store the key/value pairs 'a':"10" and 'b':"11". If the encoding tree is NULL, return an empty map.

这是编码过程的第 3 步。在此函数中，您将接受一个指向哈夫曼树（就像您在 buildEncodingTree 中创建的树一样）根节点的指针，并根据树的结构创建和返回一个哈夫曼编码映射。映射中的每个键都是一个字符，每个值都是以字符串形式表示的该字符的二进制编码。例如，如果字符" **a** "的二进制值为 **10** ，而" **b** "的二进制值为 **11** ，那么该映射应存储键/值对 "a": "10 "和 "b": "11"。如果编码树为 NULL，则返回空 map。

void encodeData(istream& input, const Map<int, string>& encodingMap, obitstream& output)

This is Step 4 of the encoding process. In this function you will read one character at a time from a given input file, and use the provided encoding map to encode each character to binary, then write the character's encoded binary bits to the given bit output bit stream. After writing the file's contents, you should write a single occurrence of the binary encoding for PSEUDO_EOF into the output so that you'll be able to identify the end of the data when decompressing the file later. You may assume that the parameters are valid: that the encoding map is valid and contains all needed data, that the input stream is readable, and that the output stream is writable. The streams are already opened and ready to be read/written. You do not need to prompt the user or open/close the files yourself.

这是编码过程的第 4 步。在此函数中，您将从给定的输入文件中一次读取一个字符，并使用提供的编码映射将每个字符编码为二进制，然后将字符的编码二进制位写入给定的比特输出比特流。写入文件内容后，应将 PSEUDO_EOF 的二进制编码的单次出现写入输出，以便在以后解压文件时能够识别数据的结束。您可以假设参数有效：编码映射有效并包含所有需要的数据，输入流可读，输出流可写。数据流已经打开，随时可以读/写。您无需提示用户或自己打开/关闭文件。

void decodeData(ibitstream& input, HuffmanNode *encodingTree, ostream& output)

This is the decoding-a-file process described previously. In this function you should do the opposite of encodeData. You read bits from the given input file one at a time, and_walk through the specified decoding tree to write the original uncompressed contents of that file to the given output stream. The streams are already opened and you do not need to prompt the user or open/close the files yourself.

这就是之前描述的文件解码过程。在这个函数中，你应该做的与 encodeData 相反。每次从给定的输入文件中读取一个比特，然后在指定的解码树中走一遍，将该文件未经压缩的原始内容写入给定的输出流。这些流已经打开，你不需要提示用户或自己打开/关闭文件。

To manually verify that your implementations of encodeData and decodeData are working correctly, use our provided test code to compress strings of your choice into a sequence of 0s and 1 s . The next page describes a header that you will add to compressed files, but in encodeData and decodeData, you should not write or read this header from the file. Instead, just use the encoding tree you're given. Worry about headers only in compress/decompress.

要手动验证 encodeData 和 decodeData 的实现是否正常，请使用我们提供的测试代码，将您选择的字符串压缩为 0 和 1 s 的序列。 下一页将介绍您将添加到压缩文件中的标头，但在 encodeData 和 decodeData 中，您不应写入或读取文件中的标头。相反，只需使用给定的编码树即可。只有在压缩/解压缩时才需要担心头文件。

The functions described above implement Huffman's algorithm, but they have one big flaw. The decoding function requires the encoding tree to be passed in as a parameter. Without the encoding tree, you don't know the mappings from bit patterns to characters, so you can't successfully decode the file.

上述函数实现了哈夫曼算法，但它们有一个很大的缺陷。解码函数需要将编码树作为参数传入。如果没有编码树，就不知道从位模式到字符的映射，因此无法成功解码文件。

We will work around this by writing the encodings into the compressed file, as a header. The idea is that when opening our compressed file later, the first several bytes will store our encoding information, and then those bytes are immediately followed by the compressed binary bits that we compressed earlier. It's actually easier to store the character frequency table, the map from Step 1 of the encoding process, and we can generate the encoding tree from that. For our **ab ab cab** example, the frequency table stores the following (the keys are shown by their ASCII integer values, such as **32** for ' ' and **97** for ' **a** ', because that is the way the map would look if you printed it out):

我们将把编码作为头写入压缩文件来解决这个问题。这样做的目的是，以后打开压缩文件时，前几个字节将存储我们的编码信息，然后紧接着这些字节的就是我们之前压缩的二进制位。实际上，存储字符频率表、编码过程步骤 1 中的映射表更为简单，我们可以从中生成编码树。以我们的 **ab ab cab** 为例，频率表存储了以下内容（键值以 ASCII 整数值表示，如 **32** 表示"'， **97** 表示" **a** "，因为如果打印出来，映射表就是这样的）：

{32 : 2, 97 : 3, 98 : 3, 99 : 1, 256 : 1}

We don't have to write the encoding header bit-by-bit; just write out normal ASCII characters for our encodings. We could come up with various ways to format the encoding text, but this would require us to carefully write code to write/read the encoding text. There's a simpler way. You already have a Map of character frequency counts from Step 1 of encoding. In C++, collections like Maps can easily be read and written to/from streams using << and >> operators. So all you need to do for your header is write your map into the bit output stream first before you start writing bits into the compressed file, and read that same map back in first later when you decompress it. The overall file is now 34 bytes: 31 for the header and 3 for the binary compressed data. Here's an attempt at a diagram: 重试 错误原因

byte 重试 错误原因	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
	'{' 重试 错误原因	'3' 重试 错误原因	'2' 重试 错误原因	':' 重试 错误原因	'2' 重试 错误原因	',' 重试 错误原因	,	'9' 重试 错误原因	'7' 重试 错误原因	':' 重试 错误原因	'3' 重试 错误原因	',' 重试 错误原因		'9' 重试 错误原因	'8' 重试 错误原因	':' 重试 错误原因	'3' 重试 错误原因
byte 重试 错误原因	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34
	',' 重试 错误原因	. ' 重试 错误原因	'9' 重试 错误原因	'9' 重试 错误原因	':' 重试 错误原因	'1' 重试 错误原因	',' 重试 错误原因		'2' 重试 错误原因	'5' 重试 错误原因	'6' 重试 错误原因	':' 重试 错误原因	'1' 重试 错误原因	'}' 重试 错误原因	*	*	*
put put. 重试 错误原因	fr ite 重试 错误原因	en (. 重试 错误原因	abl 重试 错误原因	input >> frequencyTable; input.readBit(); ... 重试 错误原因									1011		1000		/

Looking at this new rendition of the compressed file, you may be thinking, "The file is not compressed at all; it actually got larger than it was before! It went up from 9 bytes (**ab ab cab**) to 34! That's true for this contrived example. But for a larger file, the cost of the header is not so bad relative to the overall file size. There are more compact ways of storing the header, too, but they add too much challenge to this assignment, which is meant to practice trees and data structures and problem solving more than it is meant to produce a truly tight compression. 重试 错误原因

The last step is to glue all of your code together, along with code to read and write the encoding table to the file: 重试 错误原因

void compress(istream& input, obitstream& output) 重试 错误原因
This is the overall compression function. In this function you should compress the given input file into the given output file. You will take as parameters an input file that should be encoded and an output bit stream to which the compressed bits of that input file should be written. You should read the input file one character at a time, building an encoding of its contents, and write a compressed version of that input file, including a header, to the specified output file. This function should be built on top of the other encoding functions and should call them as needed. You may assume that the streams are both valid and read/writeable, but the input file might be empty. The streams are already opened and ready to be read/written. You do not need to prompt the user or open/close the files yourself.

这是一个整体压缩函数。在此函数中，您应将给定的输入文件压缩为给定的输出文件。参数包括一个需要编码的输入文件和一个输出比特流，输入文件的压缩比特将被写入输出比特流。您应逐个字符读取输入文件，对其内容进行编码，并将该输入文件的压缩版本（包括页眉）写入指定的输出文件。此函数应建立在其他编码函数之上，并在需要时调用它们。您可以假定数据流都是有效和可读/可写的，但输入文件可能是空的。数据流已经打开，随时可以读/写。您无需提示用户或自己打开/关闭文件。

void decompress(ibitstream& input, ostream& output)
This function should do the opposite of compress. It should read the bits from the given input file one at a time, including your header packed inside the start of the file, to write the original contents of that file to the file specified by the output parameter. You may assume that the streams are valid and read/writeable, but the input file might be empty. The streams are already open and ready to be used; you do not need to prompt the user or open/close files.

该函数的作用与压缩相反。它应从给定的输入文件中逐个读取比特位，包括打包在文件开头的头文件，然后将该文件的原始内容写入输出参数指定的文件。您可以假定流是有效和可读/可写的，但输入文件可能是空的。数据流已经打开并准备就绪，无需提示用户或打开/关闭文件。

void freeTree(HuffmanNode* node)
This function should free the memory associated with the tree whose root node is represented by the given pointer. You must free the root node and all nodes in its sub-trees. There should be no effect if the tree passed is NULL. If your compress or decompress function creates a Huffman tree, that function should also free the tree.

该函数将释放与树相关联的内存，给定指针代表该树的根节点。您必须释放根节点及其子树中的所有节点。如果传递的树为 NULL，则不会产生任何影响。如果您的压缩或解压缩函数创建了一个 Huffman 树，该函数也应释放该树。

Development Strategy and Hints

发展战略和提示

When writing the bit patterns to the compressed file, note that you do not write the ASCII characters ' **0** ' and ' **1′** ' (that wouldn't do much for compression!), instead the bits in the compressed form are written one-by-one using the readBit and writeBit member functions on the bitstream objects. Similarly, when you are trying to read bits from a compressed file, don't use >> or byte-based methods like get or getline; use readBit. The bits that are returned from readBit will be either 0 or 1 , but not ' **0** ' or ' **1′**.

在向压缩文件中写入比特模式时，请注意不要写入 ASCII 字符" **0** "和" **1′** "（这对压缩没有什么好处！），而是使用比特流对象上的 readBit 和 writeBit 成员函数逐个写入压缩形式的比特。同样，从压缩文件中读取比特时，不要使用 >> 或基于字节的方法（如 get 或 getline），而应使用 readBit。从 readBit 返回的位将是 0 或 1，但不是" **0** "或" **1′** "。

Work step-by-step. Get each part of the encoding program working before starting on the next one. You can test each function individually using our provided client program, even if others are blank or incomplete.

循序渐进。先让编码程序的每个部分都能正常工作，然后再开始下一个部分。您可以使用我们提供的客户端程序单独测试每个功能，即使其他功能是空白或不完整的。

Start out with small test files (two characters, ten characters, one sentence) to practice on before you start trying to compress large books of text. What sort of files do you expect Huffman to be particularly effective at compressing? On what sort of files will it be less effective? Are there files that grow instead of shrink when Huffman encoded? Consider creating sample files to test out your theories.

在开始尝试压缩大型文本文件之前，先从小型测试文件（两个字符、十个字符、一句话）开始练习。您认为哈夫曼压缩哪类文件特别有效？对哪类文件效果较差？是否有文件在进行哈夫曼编码时会不缩反增？考虑创建样本文件来测试您的理论。

Your implementation should be robust enough to compress any kind of file: text, image, video, or even one it has previously compressed. Your program probably won't be able to further squish an already compressed file (and in fact, it can get

您的程序应该足够强大，可以压缩任何类型的文件：文本、图像、视频，甚至是之前压缩过的文件。你的程序可能无法进一步压缩已经压缩过的文件（事实上，它可能会导致

larger because of header overhead) but it should be possible to compress multiple iterations, decompress the same number of iterations, and return to the original file.

但压缩多次迭代、解压缩相同次数的迭代并返回原始文件应该是可能的。

Your program only has to decompress valid files compressed by your program. You do not need to take special precautions to protect against user error such as trying to decompress a file that isn't in the proper compressed format.

你的程序只需解压由你的程序压缩的有效文件。您不需要采取特别的预防措施来防止用户出错，比如尝试解压格式不正确的文件。

See the input/output streams section for how to rewind a stream to the beginning if necessary.

如有必要，请参阅输入/输出数据流部分，了解如何将数据流倒回开头。

The operations that read and write bits are somewhat inefficient and working on a large file (100 K and more) will take some time. Don't be concerned if the reading/writing phase is slow for very large files.

读取和写入比特的操作效率较低，处理大文件（100 K 或更多）需要一些时间。如果大文件的读/写阶段很慢，也不必担心。

Note that Qt Creator puts the compressed binary files created by your code in your "build" folder. They won't show up in the normal res resource folder of your project.

请注意，Qt Creator 会将代码生成的压缩二进制文件放到 "构建 "文件夹中。它们不会显示在项目的普通 res 资源文件夹中。

Extras 额外内容

There are all sorts of fun extras you can layer on top of this assignment. Here are a few things to consider (though everything is optional):

在这项任务的基础上，您还可以添加各种有趣的额外内容。以下是一些可以考虑的事项（尽管所有事项都是可选的）：

Make the encoding table more efficient: Our implementation of the encoding table at the start of each file is not at all efficient, and for small files can take up a lot of space. Try to see if you can find a better way of encoding the data. If you're feeling up for a challenge, try looking up succinct data structures and see if you can write out the encoding tree using one bit per node and one byte per character!

提高编码表的效率：我们在每个文件开头使用的编码表效率不高，对于小文件来说可能会占用大量空间。试试看能否找到更好的数据编码方法。如果你想挑战一下，可以尝试查找简洁的数据结构，看看能否用每个节点一个比特、每个字符一个字节的方式写出编码树！

Add support for encryption in addition to encoding: Without knowledge of the encoding table, it's impossible to decode compressed files. Update the encoding table code so that it prompts for a password or uses some other technique to make it hard for Bad People to decompress the data. (There's legit extra credit opportunity here.)

除编码外，增加对加密的支持：如果不知道编码表，就无法解码压缩文件。更新编码表代码，使其提示密码或使用其他技术，让坏人难以解压数据。(这里有合法的加分机会）。

Implement a more advanced compression algorithm: Huffman encoding is a good compression algorithm, but there are much better alternatives in many cases. Try researching and implementing a more advanced algorithm, like LZW, in addition to Huffman coding. (More legitimate extra credit opportunity here as well.)

采用更先进的压缩算法：哈夫曼编码是一种很好的压缩算法，但在很多情况下还有更好的替代算法。除了哈夫曼编码外，尝试研究并实施一种更先进的算法，如 LZW。(这里还有更多合法的额外学分机会）。

Gracefully handle bad input files: The normal version of the program doesn't work very well if you feed it bogus input, such as a file that wasn't created by your own algorithm. Make your code more robust by making it able to detect whether a file is valid or invalid and react accordingly. One possible way of doing this would be to insert special bits/bytes near the start of the file that indicate a header flag or checksum. You can test to see whether these bit patterns are present, and if not, you know the file is bogus.

优雅地处理不良输入文件：如果输入的是假文件，例如不是由你自己的算法创建的文件，普通版本的程序就不能很好地工作。通过检测文件是否有效并做出相应的反应，让代码更加强大。一种可行的方法是在文件开头插入特殊的比特/字节，以指示头标志或校验和。你可以测试这些比特模式是否存在，如果不存在，你就知道文件是假的。

Anything else: If you have your own creative idea for an extra feature, ask your SL and/or the instructor about it.

其他：如果你对额外功能有自己的创意想法，请向你的 SL 和/或指导老师咨询。

Indicating that you have done extra features: If you complete any extra features, then in the comment heading on the top of your program, please list all extras that you worked on and where in the code they can be found (what functions, lines, etc. so that the grader can look at their code easily).

说明您完成了额外功能：如果您完成了任何额外功能，那么请在程序顶部的注释标题中列出您完成的所有额外功能，以及在代码中可以找到它们的位置（什么函数、行等，以便评分员可以轻松查看他们的代码）。

Submitting a program with extra features: Since we use automated testing for part of our grading process, it is important that you submit a program that conforms to the preceding spec, even if you want to add extra features. If your extras cause your program to change the output that it produces in such a way that it no longer matches the expected sample output test cases provided, you should submit your program twice: a first time without any extra features added (or with all necessary extras disabled or commented out), and a second time with the extras enabled. Please distinguish them by explaining which is which in the comment header. Our submission system saves every submission you make, so if you make multiple submissions, we will be able to view all of them; your previously submitted files will not be lost or overwritten.

提交带有额外功能的程序：由于我们在评分过程中使用自动测试，因此即使您想添加额外功能，也必须提交符合前述规范的程序。如果您的附加功能会导致程序产生的输出发生变化，从而不再符合所提供的预期输出示例测试用例，那么您应提交两次程序：第一次不添加任何附加功能（或禁用或注释掉所有必要的附加功能），第二次启用附加功能。请在注释标题中说明是哪一次，以示区别。我们的提交系统会保存您提交的每一份文件，因此如果您提交了多份文件，我们可以查看所有文件；您之前提交的文件不会丢失或被覆盖。