CSE 100: HUFFMAN CODES
Doing data compression the right way…
The Data Compression Problem

Digitize  Store  Communicate

| 11001110 | 01110000 | 01100111 | 00000011 | 11110000 | 11111111 | 01010001 | 11000111 |
| 00001011 | 11101111 | 00001000 | 01001011 | 11011110 | 00010000 | 10010111 | 10111100 |
| 00100001 | 00101111 | 01111000 | 01000010 | 01011110 | 11110000 | 01000110 | 01111100 |
| 00011011 | 11001111 | 11111110 | 11100011 | 11000001 | 00010111 | 11011011 | 01011101 |
| 11001010 | 11101111 | 10010010 | 10010101 | 11011111 | 00100100 | 00101001 | 00011010 |
| 01010011 | 00110000 | 01010100 | 01010011 | 11010100 | 01100111 | 10111010 | 01011001 |
| 01000011 | 11101111 | 00010111 | 00010000 | 10011000 | 00111011 | 00010000 | 11011000 |
| 10001100 | 10010110 | 00001010 | 00010000 | 11001100 | 11001110 | 01110011 | 01011101 |
| 00011010 | 01101101 | 00000001 | 11000100 | 11001100 | 01110111 | 01101011 | 11011100 |
RIGHT HO, JEEVES

By

P. G. WODEHOUSE

1922
To

RAYMOND NEEDHAM, K.C.
WITH AFFECTION AND ADMIRATION

"Jeeves," I said, "may I speak frankly?"
"Certainly, sir."
"What I have to say may wound you."
"Not at all, sir."
"Well, then——"
No—wait. Hold the line a minute. I've gone off the rails.

I don't know if you have had the same experience, but the snag I always come up against when I'm telling a story is this dashed difficult problem of where to begin it. It's a thing you don't want to go wrong over, because one false step and you're sunk. I mean, if you fool about too long at the start, trying to establish atmosphere, as they call it, and all that sort of rot, you fail to grip and the customers walk out on you.

Get off the mark, on the other hand, like a scalded cat, and your public is at a loss. It simply raises its eyebrows, and can't make out what you're talking about.

And in opening my report of the complex case of Gussie Fink-Nottle, Madeline Bassett, my Cousin Angela, my Aunt Dahlia, my Uncle Thomas, young Tuppy Glossop and the cook, Anatole, with the above spot of dialogue, I see that I have made the second of these two floaters.

I shall have to hark back a bit. And taking it for all in all and weighing this against that, I suppose the affair may be said to have had its inception, if inception is the word I want, with that visit of mine to Cannes. If I hadn't gone to Cannes, I shouldn't have met the Bassett or bought that white mess jacket, and Angela wouldn't have met her shark, and Aunt Dahlia wouldn't have played baccarat.

Yes, most decidedly, Cannes was the point d'appui.

Right ho, then. Let me marshal my facts.

I went to Cannes—leaving Jeeves behind, he having intimated that he did not wish to miss Ascot—round about the beginning of June. With me travelled my Aunt Dahlia and her daughter Angela. Tuppy Glossop, Angela's betrothed, was to have been of the party, but at the last moment couldn't get away. Uncle Tom, Aunt Dahlia's husband, remained at home, because he can't stick the South of France at any price.
How to represent in bits?
- Encoding with ASCII
- Symbols and dictionaries

How many bits do we need?

The ASCII table

<table>
<thead>
<tr>
<th>Dec</th>
<th>Hex</th>
<th>ASCII</th>
<th>Oct</th>
<th>Chr</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>30</td>
<td>NULL</td>
<td>00</td>
<td>Space</td>
</tr>
<tr>
<td>1</td>
<td>31</td>
<td>Start of Header</td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>End of Text</td>
<td>02</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>33</td>
<td>End of Transmission</td>
<td>03</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>34</td>
<td>End of Transmission</td>
<td>04</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>35</td>
<td>End of Transmission</td>
<td>05</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>36</td>
<td>End of Transmission</td>
<td>06</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>37</td>
<td>End of Transmission</td>
<td>07</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>38</td>
<td>End of Transmission</td>
<td>08</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>39</td>
<td>End of Transmission</td>
<td>09</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>3A</td>
<td>End of Transmission</td>
<td>0A</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>3B</td>
<td>End of Transmission</td>
<td>0B</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>3C</td>
<td>End of Transmission</td>
<td>0C</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>3D</td>
<td>End of Transmission</td>
<td>0D</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>3E</td>
<td>End of Transmission</td>
<td>0E</td>
<td></td>
</tr>
<tr>
<td>15</td>
<td>3F</td>
<td>End of Transmission</td>
<td>0F</td>
<td></td>
</tr>
</tbody>
</table>
Fixed length encoding

- Fixed length: each symbol is represented using a fixed number of bits
- For example if the symbols were ‘s’, ‘p’, ’a’, ‘m’ we might define the following encoding:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Code word</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>00</td>
</tr>
<tr>
<td>p</td>
<td>01</td>
</tr>
<tr>
<td>a</td>
<td>10</td>
</tr>
<tr>
<td>m</td>
<td>11</td>
</tr>
</tbody>
</table>

For a dictionary consisting of M symbols, what is the minimum number of bits needed to encode each symbol (assume fixed length binary codes) ?

A. $2^M$  B. M  C. $M/2$  D. $\text{ceil}(\log_2 M)$  E. None of these
Binary codes as Binary Trees

Symbols are leaf nodes
Root to leaf node gives the codeword for each symbol

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>00</td>
</tr>
<tr>
<td>p</td>
<td>01</td>
</tr>
<tr>
<td>a</td>
<td>10</td>
</tr>
<tr>
<td>m</td>
<td>11</td>
</tr>
</tbody>
</table>
Encoding with Binary Trees

Given the above binary tree, encode the string “papa”
Encoding a symbol- think implementation!

A very bad way is to start at the root and search down the tree until you find the symbol you are trying to encode.
A much better way is to maintain a list of leaves and then to traverse the tree to the root (and then reverse the code).

```cpp
vector<HCNode*> leaves;
...
leaves = vector<HCNode*>(256, (HCNode*)0);
```
Traversing a list, with recursion

class LNode {
    int data;
    LNode* next;
}

void traverse(LNode* n) {
    // 1
    if (n == nullptr) return;
    // 2
    traverse(n->next);
    // 3
}

Where should I put the line to print n->data to print the list in reverse order?

std::cout << n->data << std::endl;

A. 1  B. 2  C. 3
A much better way is to maintain a list of leaves and then to traverse the tree to the root. If you use recursion, there’s no need to reverse the list!

Use recursion to easily write a symbol’s code in the correct order!

```cpp
vector<HCNode*> leaves;
...
leaves = vector<HCNode*>(256, (HCNode*)0);
```
Decoding on binary trees

Decode the bitstream 110101001100 using the given binary tree
A. scam
B. mork
C. rock
D. korp
Is there an alternative to fixed length encoding?

• Can we go beyond fixed length encoding?
• What if certain symbols appeared more often than others?
Is code B better than code A?
A. Yes
B. No
C. Depends
Comparing encoding schemes

Average length (code A) = 2 bits/symbol
Average length (code B) = 0.6 *1 + 0.2 *1 + 0.1* 2+ 0.1*2
= 1.2 bits/symbol
Decoding variable length codes

Symbol Frequency
s 0.6
p 0.2
a 0.1
m 0.1

Symbol Codeword
s 00
p 01
a 10
m 11

Decode the binary pattern 0110 using Code B?
A. spa
B. sms
C. Not enough information to decode
Variable length codes

Variable length codes have to necessarily be prefix codes for correct decoding.

A **prefix code** is one where no symbol’s codeword is a prefix of another.

**Code A**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>00</td>
</tr>
<tr>
<td>p</td>
<td>01</td>
</tr>
<tr>
<td>a</td>
<td>10</td>
</tr>
<tr>
<td>m</td>
<td>11</td>
</tr>
</tbody>
</table>

**Code B**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0</td>
</tr>
<tr>
<td>p</td>
<td>1</td>
</tr>
<tr>
<td>a</td>
<td>10</td>
</tr>
<tr>
<td>m</td>
<td>11</td>
</tr>
</tbody>
</table>

Code B is not a prefix code.
Variable length codes

Is code C better than code A and code B? (Why or why not?)
A. Yes
B. No

<table>
<thead>
<tr>
<th>Code A</th>
<th>Symbol</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>00</td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>01</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>m</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code B</th>
<th>Symbol</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>m</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code C</th>
<th>Symbol</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>p</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>110</td>
<td></td>
</tr>
<tr>
<td>m</td>
<td>111</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0.6</td>
</tr>
<tr>
<td>p</td>
<td>0.2</td>
</tr>
<tr>
<td>a</td>
<td>0.1</td>
</tr>
<tr>
<td>m</td>
<td>0.1</td>
</tr>
</tbody>
</table>
Variable length codes

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>00</td>
</tr>
<tr>
<td>p</td>
<td>01</td>
</tr>
<tr>
<td>a</td>
<td>10</td>
</tr>
<tr>
<td>m</td>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0</td>
</tr>
<tr>
<td>p</td>
<td>1</td>
</tr>
<tr>
<td>a</td>
<td>10</td>
</tr>
<tr>
<td>m</td>
<td>11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0</td>
</tr>
<tr>
<td>p</td>
<td>10</td>
</tr>
<tr>
<td>a</td>
<td>110</td>
</tr>
<tr>
<td>m</td>
<td>111</td>
</tr>
</tbody>
</table>

Symbol | Frequency |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0.6</td>
</tr>
<tr>
<td>p</td>
<td>0.2</td>
</tr>
<tr>
<td>a</td>
<td>0.1</td>
</tr>
<tr>
<td>m</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Average length (code A) = 2 bits/symbol
Average length (code B) = 0.6 *1 +0.2 *1 + 0.1* 2+ 0.1*2
                            = 1.2 bits/symbol
Average length (code C) = 0.6 *1 +0.2 *2 + 0.1* 3+ 0.1*3
                            = 1.6 bits/symbol
What is the advantage of thinking of codes as trees?

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>00</td>
</tr>
<tr>
<td>p</td>
<td>01</td>
</tr>
<tr>
<td>a</td>
<td>10</td>
</tr>
<tr>
<td>m</td>
<td>11</td>
</tr>
</tbody>
</table>

**Code A**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0</td>
</tr>
<tr>
<td>p</td>
<td>1</td>
</tr>
<tr>
<td>a</td>
<td>10</td>
</tr>
<tr>
<td>m</td>
<td>11</td>
</tr>
</tbody>
</table>

**Code B**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Codeword</th>
</tr>
</thead>
<tbody>
<tr>
<td>s</td>
<td>0</td>
</tr>
<tr>
<td>p</td>
<td>10</td>
</tr>
<tr>
<td>a</td>
<td>110</td>
</tr>
<tr>
<td>m</td>
<td>111</td>
</tr>
</tbody>
</table>

**Code C**
Problem Definition

Given a frequency distribution over M symbols, find the optimal prefix binary code i.e. one that minimizes the average code length

In short:
I give you frequencies, you give me the best tree!

Symbol Frequency
s 0.6
p 0.2
a 0.1
m 0.1

Huffman coding is one of the fundamental ideas that people in computer science and data communications are using all the time - Donald Knuth
Problem Definition (reworded for trees)

Input: The frequency \( (p_i) \) of occurrence of each symbol \( (S_i) \)

Output: Binary tree \( T \) that minimizes the following objective function:

\[
L(T) = \sum_{i=1:N} p_i \cdot \text{Depth}(S_i \text{ in } T)
\]

Solution: Huffman Codes
Huffman’s algorithm: Bottom up construction

• Build the tree from the bottom up!
• Start with a forest of trees, all with just one node

A: 6; B: 4; C: 4; D: 0; E: 0; F: 0; G: 1; H: 2
Huffman’s algorithm: Bottom up construction

- Build the tree from the bottom up!
- Start with a forest of trees, all with just one node
- Choose the two smallest trees in the forest and merge them

A: 6; B: 4; C: 4; D: 0; E: 0; F: 0; G: 1; H: 2
Huffman's algorithm: Bottom up construction

- Build the tree from the bottom up!
- Start with a forest of trees, all with just one node
- Choose the two smallest trees in the forest and merge them

A: 6; B: 4; C: 4; D: 0; E: 0; F: 0; G: 1; H: 2
Huffman’s algorithm: Bottom up construction

- Build the tree from the bottom up!
- Start with a forest of trees, all with just one node
- Choose the two smallest trees in the forest and merge them
- Repeat until all nodes are in the tree
Huffman’s algorithm: Bottom up construction

- Build the tree from the bottom up!
- Start with a forest of trees, all with just one node
- Choose the two smallest trees in the forest and merge them
- Repeat until all nodes are in the tree
Huffman’s algorithm: Bottom up construction

- Build the tree from the bottom up!
- Start with a forest of trees, all with just one node
- Choose the two smallest trees in the forest and merge them
- Repeat until all nodes are in the tree
PA3: encoding/decoding

ENCODING:
1. Scan text file to compute frequencies
2. Build Huffman Tree
3. Encode: Find code for every symbol (letter)
4. Create new compressed file by saving the entire code at the top of the file followed by the code for each symbol (letter) in the file

DECODING:
1. Read the file header (which contains the code) to recreate the tree
2. Decode each letter by reading the file and using the tree

This is the logical flow of what needs to happen, it need not be the way you develop the code: Brainstorm in class and in section to see how you can design in a modular way
PA3: encoding/decoding

ENCODING:
1. Scan text file to compute frequencies
2. Build Huffman Tree
3. Find code for every symbol (letter)
4. Create new compressed file by saving the entire code at the top of the file followed by the code for each symbol (letter) in the file

DECODING:
1. Read the file header (which contains the code) to recreate the tree
2. Decode each letter by reading the file and using the tree
Writing the compressed file

In the reference solution, the header is just 256 ints in a row, these are the counts, one for each byte value.

This header takes 1024 bytes of space. \((4\times256)\)

You must beat this for your final submission.

<table>
<thead>
<tr>
<th>Header (some way to reconstruct the HCTree)</th>
<th>Encoded data (bits)</th>
</tr>
</thead>
</table>
PA3: encoding/decoding

ENCODING:
1. Scan text file to compute frequencies
2. Build Huffman Tree
3. Find code for every symbol (letter)
4. Create new compressed file by saving the entire code at the top of the file followed by the code for each symbol (letter) in the file

DECODING:
1. Read the file header (which contains the code) to recreate the tree
2. Decode each letter by reading the file and using the tree
Building the tree: Huffman’s algorithm

0. Determine the count of each symbol in the input message.

1. Create a forest of single-node trees containing symbols and counts for each non-zero-count symbol.

2. Loop while there is more than 1 tree in the forest:
   2a. Remove the two lowest count trees
   2b. Combine these two trees into a new tree (summing their counts).
   2c. Insert this new tree in the forest, and go to 2.

3. Return the one tree in the forest as the Huffman code tree.

You know how to create a tree. But how do you maintain the forest?
What is a Heap?

Think of a Heap as a binary tree that is as complete as possible and satisfies the following property:

At every node $x$

$$\text{Key}[x] \leq \text{Key}[\text{children of } x]$$

So the root has the ______ value
The suitability of Heap for our problem

- In the Huffman problem we are doing repeated inserts and extract-min!
- Perfect setting to use a Heap data structure.
- The C++ STL container class: priority_queue has a Heap implementation.
- Priority Queue and Heap are synonymous
A C++ priority_queue is a generic container, and can hold any kind of thing as specified with a template parameter when it is created: for example HCNode, or pointers to HCNode, etc.

```cpp
#include <queue>
std::priority_queue<HCNode> p;
```

- You can extract object of highest priority in O(log N)
- To determine priority: objects in a priority queue must be comparable to each other
- By default, a priority_queue<T> uses operator< defined for objects of type T:
  - if a < b, b is taken to have higher priority than a
#ifndef HCNODE_HPP
#define HCNODE_HPP

class HCNode { 

public:
    HCNode* parent; // pointer to parent; null if root
    HCNode* child0; // pointer to "0" child; null if leaf
    HCNode* child1; // pointer to "1" child; null if leaf
    unsigned char symb; // symbol
    int count; // count/frequency of symbols in subtree

    // for less-than comparisons between HCNodes
    bool operator<(HCNode const & ) const;
};

#endif
#include HCNODE_HPP
/** Compare this HCNode and other for priority ordering.
 * Smaller count means higher priority.
 */
bool HCNode::operator<(HCNode const & other) const {
    // if counts are different, just compare counts
    return count > other.count;
};

What is wrong with this implementation?
A. Nothing
B. It is non-deterministic (in our algorithm)
C. It returns the opposite of the desired value for our purpose
#include HCNODE_HPP
/** Compare this HCNode and other for priority ordering.
 * Smaller count means higher priority.
 * Use node symbol for deterministic tiebreaking
 */

bool HCNode::operator<(HCNode const & other) const {
    // if counts are different, just compare counts
    if(count != other.count) return count > other.count;
    // counts are equal. use symbol value to break tie.
    // (for this to work, internal HCNodes
    // must have symb set.)
    return symb < other.symb;
};
Using < to compare nodes

• Consider this context:
  \begin{verbatim}
  HCNode n1, n2, n3, n4;
  n1.count = 100;  n1.symbol = 'A';
  n2.count = 200;  n2.symbol = 'B';
  n3.count = 100;  n3.symbol = 'C';
  n4.count = 100;  n4.symbol = 'A';
  \end{verbatim}

• Now what is the value of these expressions?
  \begin{align*}
  & n1 < n2 \quad \text{A. true} \\
  & n2 < n1 \quad \text{B. false} \\
  & n2 < n3 \\
  & n1 < n3 \\
  & n3 < n1 \\
  & n1 < n4
  \end{align*}
Using `std::priority_queue` in Huffman’s algorithm

- If you create an STL container such as `priority_queue` to hold `HCNode` objects:

  ```cpp
  #include <queue>
  std::priority_queue<HCNode> pq;
  ```

- ... then adding an `HCNode` object to the `priority_queue`:
  ```cpp
  HCNode n;
  pq.push(n);
  ```

- ... actually creates a copy of the `HCNode`, and adds the copy to the queue. You probably don’t want that. Instead, set up the container to hold pointers to `HCNode` objects:
  ```cpp
  std::priority_queue<HCNode*> pq;
  HCNode* p = new HCNode();
  pq.push(p);
  ```
Using std::priority_queue in Huffman’s

Instead, set up the container to hold pointers to HCNode objects:

```cpp
std::priority_queue<HCNode*> pq;
HCNode* p = new HCNode();
pq.push(p);
```

What is the problem with the above approach?

A. Since the priority queue is storing copies of HCNode objects, we have a memory leak

B. The nodes in the priority queue cannot be correctly compared

C. Adds a copy of the pointer to the node into the priority queue

D. The node is created on the run time stack rather than the heap
Using `std::priority_queue` in Huffman’s algorithm

Instead, set up the container to hold pointers to HCNode objects:

```cpp
std::priority_queue<HCNode*> pq;
HCNode* p = new HCNode();
pq.push(p);
```

What is the problem with the above approach?

- our operator< is a member function of the HCNode class.
  It is not defined for pointers to HCNodes. What to do?
std::priority_queue template arguments

• The template for priority_queue takes 3 arguments:

• The template for priority_queue takes 3 arguments:

| template < class T, class Container = vector<T>, |
| class Compare = less<typename Container::value_type> > class priority_queue; |

• The first is the type of the elements contained in the queue.

• If it is the only template argument used, the remaining 2 get their default values:
  • a vector<T> is used as the internal store for the queue,
  • less a class that provides priority comparisons

• Okay to use vector container, but we want to tell the priority_queue to first dereference the HCNode pointers it contains, and then apply operator<

• How to do that? We need to provide the priority_queue with a Compare class
Defining a “comparison class”

• The documentation says of the third template argument:
  • Compare: Comparison class: A class such that the expression comp(a,b), where comp is an object of this class and a and b are elements of the container, returns true if a is to be placed earlier than b in a strict weak ordering operation. This can be a class implementing a function call operator...

Here’s how to define a class implementing the function call operator operator() that performs the required comparison:

class HCNodePtrComp {
    bool operator()(HCNode* & lhs, HCNode* & rhs) const {
        // dereference the pointers and use operator<
        return *lhs < *rhs;
    }
};

Now, create the priority_queue as:
std::priority_queue<HCNode*,std::vector<HCNode*>,HCNodePtrComp> pq;

and priority comparisons will be done as appropriate.
One more piece

• When you create a new node in the forest, what character should it hold (beyond holding the sum of their children)?
  A. The character ‘0’
  B. The alphabetically smaller character of its children
  C. The alphabetically larger character of its children
  D. It doesn’t matter
PA3: encoding/decoding

ENCODING:
1. Scan text file to compute frequencies
2. Build Huffman Tree
3. Find code for every symbol (letter)
4. Create new compressed file by saving the entire code at the top of the file followed by the code for each symbol (letter) in the file

DECODING:
1. Read the file header (which contains the code) to recreate the tree
2. Decode each letter by reading the file and using the tree

These require file I/O. Do Wednesdays reading to figure this out. DON’T WAIT until after Wednesday.
PA 3 Implementation strategy

- Implement Huffman tree \texttt{build()} method
  - \texttt{HCNode.cpp} and \texttt{HCTree.cpp}
- Write verification code to check that you can construct simple Huffman trees correctly
  - Use small inputs that you can verify by hand
  - Output codes as strings of 1s and 0s (char)
- Write the encoder method and driver program
  - Test with simple inputs that you can verify by hand and output the encoded input as character strings of 1s ad 0s
- Add binary I/O
  - Write implementations of \texttt{BitInputStream} and \texttt{BitOutputStream} that write/read the compressed file as a text file (retain the ability to output in ASCII, it may come in handy)
- Compress/decompress a small file (100 bytes)
- Decompression should map the encoded input back to the original input