Huffman Encoding

Encoding Information

- encryption encodes one string into another in order to achieve privacy and authenticity
- compression encodes one string into another in order to reduce the size of the message

Fixed Length Codes

- codes like ASCII provide fixed length (8-bit) representations for each character — enough for 256 distinct characters
- a string of n characters will occupy 8n bits
- bit sequences are easily read, eight bits at a time

Variable Length Codes

- codes like Morse represent each character by a sequence of bits — a character's bit length is inversely proportional to its frequency of occurrence
- 'e' is the most common letter in english, so its code is a single bit long
- to read Morse 'bit' strings, 'pauses' are inserted between characters — thus, Morse is not truly binary

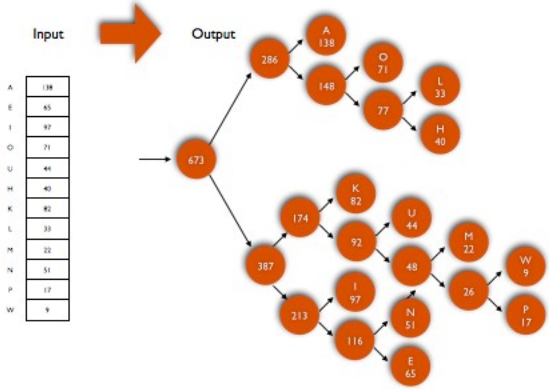
Prefix Codes

- a prefix code is a variable length code with no 'end-of-character' encoding
- no character's code is the prefix of another character's code — thus, when a bit sequence is read that completes a character, the next bit must belong to the next character

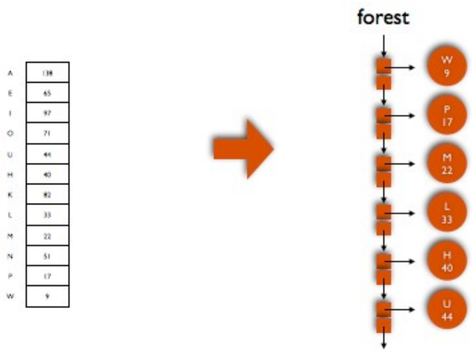
Huffman Encoding: Three Algorithms

- create the optimal code from a frequency table — represented as a binary tree (the Huffman tree) and an array of bit strings indexed by characters
- encode a character string into bit string with that optimal code
- decode a bit string into the character string

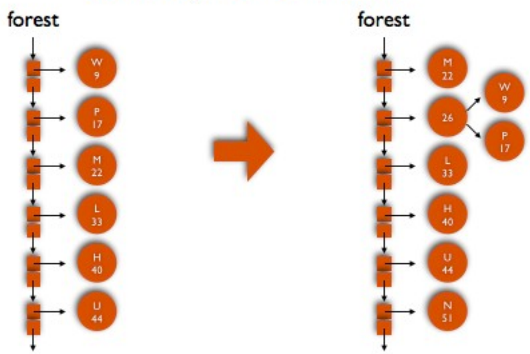
Building the Huffman Tree



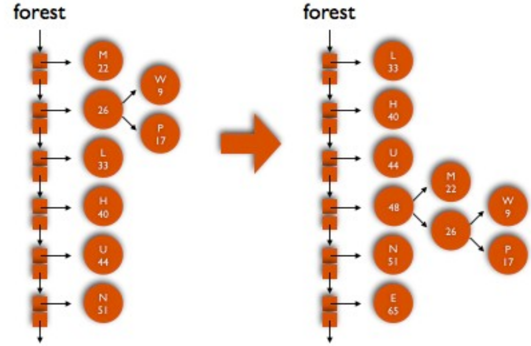
Step 1: Build a Forest of Trees



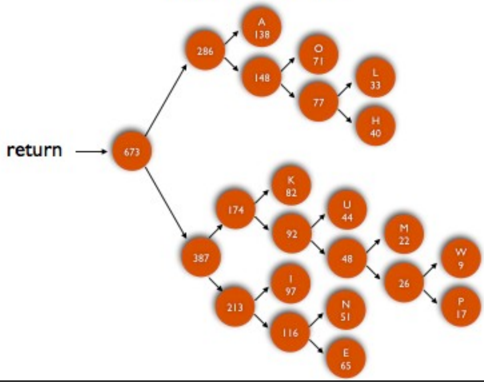
Step 2: Repeatedly Remove Two Smallest, Combine, and Insert Result



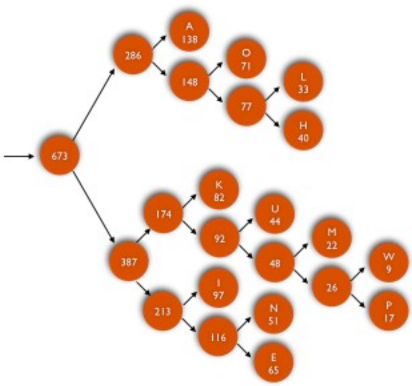
Step 2: Repeatedly Remove Two Smallest, Combine, and Insert Result



Step 3: When Forest Length is One, Return the Tree



The Code Table



A	11
E	0000
I	001
O	101
U	0101
H	1000
K	011
L	1001
M	01001
N	0001
P	010000
W	010001

Encoding Algorithm

simply look up bit string for each character in table

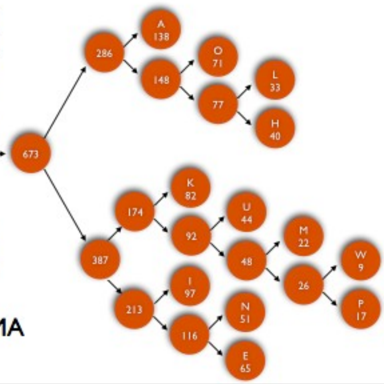
HANAUMA →

10001100011101010100111

A	11
E	0000
I	001
O	101
U	0101
H	1000
K	011
L	1001
M	01001
N	0001
P	010000
W	010001

Decoding Algorithm

start at root, 0 for left, 1 for right, until leaf found; output character at leaf; start at root again



1000110001110101010011 1

10001100011
101010100111

→ HANAUMA

```
public class HuffmanData implements Comparable {
    private int frequency;
    private char letter;

    public HuffmanData (char c, int f) {
        this.letter = c;
        this.frequency = f;
    }

    public int compareTo (Object d) {
        return ((Integer) this.frequency).compareTo(
            (Integer) ((HuffmanData) d).frequency);
    }

    public String toString () {
        return String.valueOf(this.letter);
    }

    public int frequency () { return this.frequency; }
    public char letter () { return this.letter; }

}
```

```
public final class HuffmanTree
       extends BinaryTree implements Comparable {
    private String[] table;
    private static final int NUM_CHARS = 256;
    private static final HuffmanTree NIL = new HuffmanTree();

    private HuffmanTree () { }

    private HuffmanTree (char c, int f) {
        super(new HuffmanData(c, f), NIL, NIL);
    }

    private HuffmanTree (HuffmanTree left, HuffmanTree right) {
        super(new HuffmanData((char) 0,
              left.frequency()+right.frequency()),
              left, right);
    }
.
.
.
}
```

```
public final class HuffmanTree
       extends BinaryTree implements Comparable {
.
.
.
    public HuffmanTree (String s) {
        Queue forest = HuffmanTree.analyze(s);

        while (forest.length() > 1)
            forest.enter(new HuffmanTree(
                                   (HuffmanTree) forest.leave(),
                                   (HuffmanTree) forest.leave()));

        HuffmanTree h = (HuffmanTree) forest.leave();

        this.data = h.data();
        this.left = h.left();
        this.right = h.right();
        this.table = new String[NUM_CHARS];
        this.buildCodeTable(this,"");
    }
.
.
.
}
```

```
public final class HuffmanTree
       extends BinaryTree implements Comparable {
.
.
.
    public int compareTo (Object d) {
        return this.data().compareTo(((HuffmanTree) d).data());
    }

    public boolean isEmpty () {
        return this == HuffmanTree.NIL;
    }

    private boolean isLeaf () {
        return this.left().isEmpty();
    }
.
.
.
}
```

```java
public final class HuffmanTree
       extends BinaryTree implements Comparable {
.
.
.
    private int frequency () {
        return ((HuffmanData) data()).frequency();
    }

    private char letter () {
        return ((HuffmanData) data()).letter();
    }

    public HuffmanTree left () {
        return (HuffmanTree) this.left;
    }

    public HuffmanTree right () {
        return (HuffmanTree) this.right;
    }
.
.
.
}
```

```java
public final class HuffmanTree
       extends BinaryTree implements Comparable {
.
.
.
    private static PriorityQueue analyze (String s) {
        PriorityQueue forest = new PriorityQueue(NUM_CHARS, false);
        int[] freq = new int[NUM_CHARS];

        for (int i = 0; i < s.length(); i++) freq[s.charAt(i)]++;

        for (char c = 0; c < NUM_CHARS; c++)
            if (freq[c] != 0)
                forest.enter(new HuffmanTree(c, freq[c]));

        return forest;
    }
.
.
.
}
```

```java
public final class HuffmanTree
       extends BinaryTree implements Comparable {
.
.
.
    private void buildCodeTable (HuffmanTree h, String s) {
        if (h.isLeaf()) this.table[h.letter()] = s;
        else {
            buildCodeTable(h.left(), s + '0');
            buildCodeTable(h.right(), s + '1');
        }
    }
.
.
.
}
```

```java
public final class HuffmanTree
       extends BinaryTree implements Comparable {
.
.
.
    public String encode (String s) {
        String result = "";

        for (int i = 0; i < s.length(); i++)
           result += this.table[s.charAt(i)];

        return result;
    }
.
.
.
}
```

```java
public final class HuffmanTree
       extends BinaryTree implements Comparable {
.
.
.
    public String decode (String s) {
        HuffmanTree p = this;
        String result = "";

        for (int i = 0; i < s.length(); i++) {
            p = ((s.charAt(i) == '0') ? p.left() : p.right());
            if (p.isLeaf()) {
                result += p.letter();
                p = this;
            }
        }

        return result;
    }
.
.
.
}
```

```java
public class Huffman {
    public static void main (String[] arg) {
        try {
            if (arg[0].charAt(1) != 'e' ||
                arg[0].charAt(1) != 'd')
                throw new IllegalArgumentException(
 "Usage -- Huffman [ -e | -d ] frequency-file < input > output");

            String file = "";
            String line;
            HuffmanTree h = new HuffmanTree(IO.readFile(arg[1]));

            while ((line = IO.stdin.readLine()) != null)
                file += line + "\n";

            if (arg[0].charAt(1) == 'e')
                IO.stdout.print(h.encode(file));
            else
                IO.stdout.print(h.decode(file));

        } catch (Exception e) {
            IO.stderr.println(e.getMessage());
        }
    }
}
```