Data Compression

Reference : Chapter 6.2 plus complementary material in Huffman codes.

Last Time : Buffer Management

Today

- Introduction to Data Compression
- Techniques for Data Compression
  1) Compact Notation
  2) Run-length Encoding
  3) Huffman Code

Next Time : Data Compression: Lempel-Ziv Code.

Data Compression = Encoding the information in a file in such a way that it takes less space.

Techniques for Data Compression:

1. **Using Compact Notation**

   Ex: File with fields: lastname, province, postal code, etc.
   Province field uses 2 bytes (e.g. 'ON', 'BC') but there are only 13 provinces and territories which could be encoded by using only 4 bits (compact notation).

   16 bits are encoded by 4 bits (12 bits were redundant, i.e. added no extra information)

   Disadvantages:

   - The field "province" becomes unreadable by humans.
   - Time is spent encoding ('ON' $\rightarrow$ 0001) and decoding (0001 $\rightarrow$ 'ON').
   - It increases the complexity of software.

2. **Run-length Encoding**

   Good for files in which sequences of the same byte may be frequent.

   Example: Figure 6.1 in page 205 of the textbook: image of the sky.

   - A pixel is represented by 8 bits.
   - Background is represented by the pixel value 0.

   The idea is to avoid repeating, say, 200 bytes equal to 0 and represent it by (0, 200).

If the same value occurs more than once in succession, substitute by 3 bytes:

- a special character - run length code indicator (use 1111 1111 or FF in hexadecimal notation)
- the pixel value that is repeated (FF is not a valid pixel anymore)
- the number of times the value is repeated (up to 256 times)

Encode the following sequence of Hexadecimal bytes:

```
22  23  24  24  24  24  24  24  24  25
26  26  26  26  26  26  25  24
```

Run-length encoding:

```
22  23  FF  24  07  25  FF  26  06  25  24
```

18 bytes reduced to 11.

3. **Variable Length Codes and Huffman Code**

Example of a variable length code:

Morse Code (two symbols associated to each letter)

```
A     ._
B     _...
...
E     .
F     .._.
...
T     _
U     .._
...
```

Since E and T are the most frequent letters, they are associated to the shortest codes (. and - respectively)

Huffman Code is a variable length code, but unlike Morse Code the encoding depends on the frequency of letters occurring in the data set.

Example of Huffman Code:

Suppose the file content is:

I		A	M		S	A	M	M	Y

Total: 10 characters

Letter	A	I	M	S	Y	/b
Frequency	2	1	3	1	1	2
Code	00	1010	11	1011	100	01

* Huffman Code is a prefix code: no codeword is a prefix of any other. (we are representing the space as "/b")

Encoded message

`1010010011011011001111100`

25 bits rather than 80 bits (10 bytes)

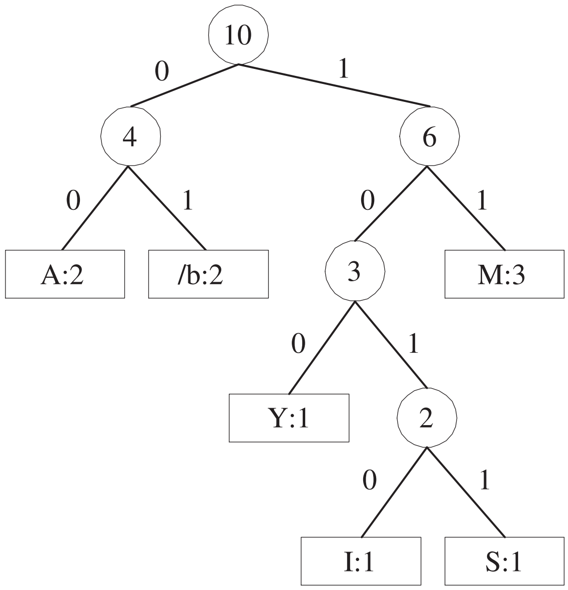
Huffman Tree (for easy decoding)

Consider the encoded message:

`101001001101...`

- Interpret the 0's as "go left" and the 1's as "go right".
- A **codeword** for a character corresponds to the path from the root of the Huffman tree to the leaf containing the character.

Following the labeled edges in the Huffman tree we decode the above message.




```
1010    leads us to I
01      leads us to /b
00      leads us to A
11      leads us to M
01      leads us to /b
etc.
```

Properties of Huffman Tree:

- Every internal node has 2 children
- Smaller frequencies are further away from the tree
- The 2 smallest frequencies are siblings
- The number of bits required to encode the file is minimized:

$$B(T) = \sum_{(c \in C)} f(c).d_T(c)$$

Where:
$B(T)$ = number of bits needed to encode the file using tree T,

$f(c)$ = frequency of character c,
$d_T(c)$ = length of the codeword for character c.

In our example:

$B(T) = 2 \times 2 + 1 \times 4 + 3 \times 2 + 1 \times 4 + 1 \times 3 + 2 \times 2 = 25$

What is the average number of bits per encoded letter ?

Average number of bits per letter = B(T)/total number of characters = 25/10 = 2.5

The way Huffman Tree is constructed guarantees that B(T) is as small as possible.

How the Huffman Tree is constructed ?

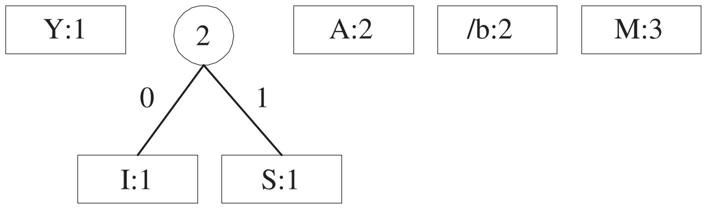
The algorithm employs a **Greedy Method** that always merges the subtrees of smallest weights forming a new subtree whose root has the sum of the weight of its children.

The algorithm in action:

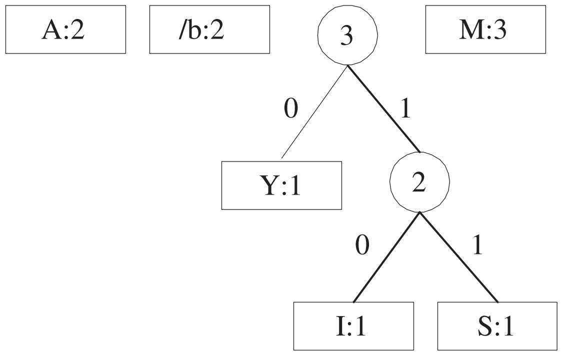
Subtrees: $\boxed{\text{I:1}}$ $\boxed{\text{S:1}}$ $\boxed{\text{Y:1}}$ $\boxed{\text{A:2}}$ $\boxed{\text{/b:2}}$ $\boxed{\text{M:3}}$

Merge the two subtrees of smallest weight (break ties arbitrarily) :

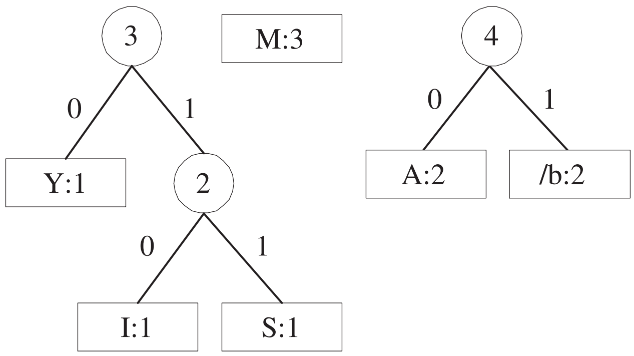
Subtrees:



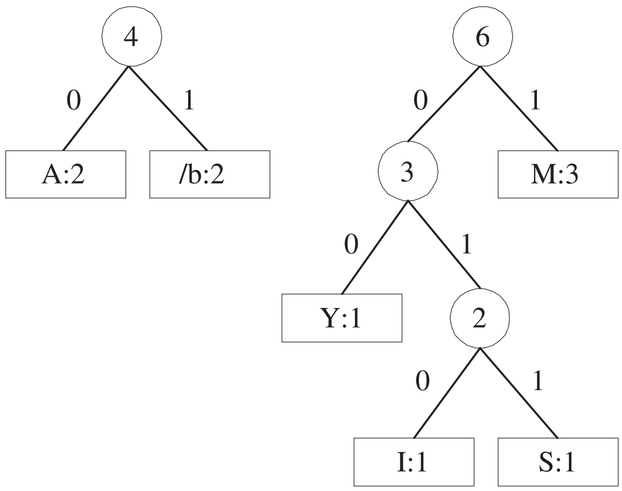
Subtrees:



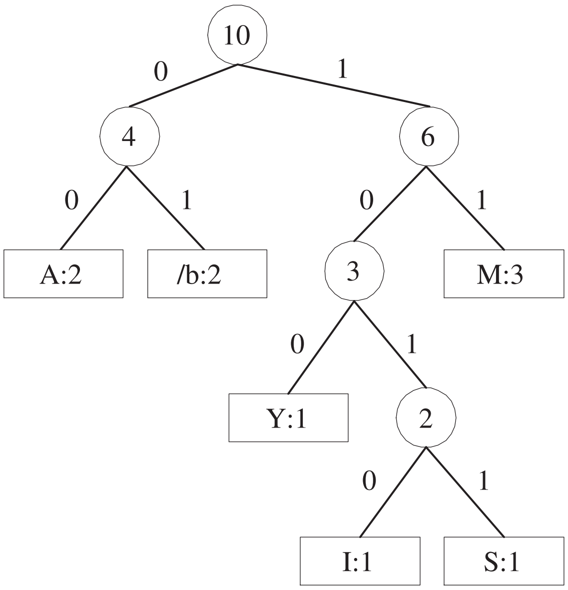
Subtrees:



Subtrees:



Final Tree:



Pseudo-Code for Huffman Algorithm:

A priority queue `Q` is used to identify the smallest-weight subtrees to merge.

A priority queue provides the following operations:

- `Q.insert(x)`: insert `x` to `Q`
- `Q.minimum()`: returns element of smallest key
- `Q.extract-min()`: removes and returns the element with smallest key

Ex:
Heaps: Each of the three operations can be done in $O(\log_n)$
Arrays: Each of the three operations can be done in O(n)

Pseudo-Code: `Huffman`
Input: characters and their frequencies
`(c1, f[c1]), (c2, f[c2]), ..., (cn, f[cn])`
Output: returns the Huffman Tree

```
{  Make heap Q using c1, c2, ..., cn;

   For i = 1 to n - 1 do {
       z = allocate new node;
       l = Q.extract-min();
       r = Q.extract-min();
       z.left = l;
       z.right = r;
       f[z] = f[r] + f[l];
       Q.insert(z);
   }
   Return Q.extract-min();
}
```

What is the running time of this algorithm if the priority queue is implemented as:

A) Heap (Array Heap)
- Build heap $O(n.\log_n)$ or $O(n)$
- Extract-min and insert takes $O(\log_n)$
- Loop iterates n-1 times
total time: $O(n.\log_n)$

B) List
- Build heap takes $O(n)$
- Extract-min and insert takes $O(n)$
- Loop iterates n-1 times
total time: $O(n^2)$

- **Pack** and **unpack** commands in Unix use Huffman Codes byte-by-byte.
- They achieve 25 - 40% reduction on text files, but is not so good for binary files that have more uniform distribution of values.