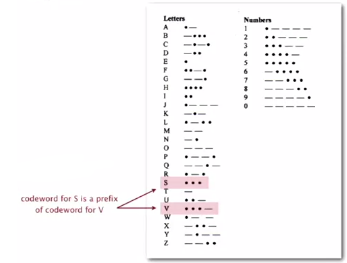
Huffman Compression



Variable length codes: use different number of bits to encode   
different chars

Example: Morse code ( \* \* \* -- -- -- \* \* \* )

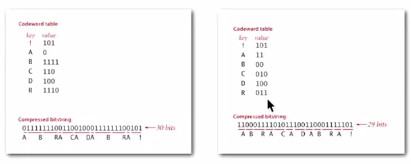
Issue: Ambiguity- the above could be…

* SOS ?
* V7 ?
* IAMIE ?
* EEWNI ?

In practice: use medium gap to separate codewords

How to avoid ambiguity: Ensure that no codeword is a prefix of another

* Example 1: fixed-length code
* Example 2: append special stop char to each codeword
* Example 3: generate prefix-free code (no code is the prefix of another)



How to represent prefix-free code -> Use a binary trie:

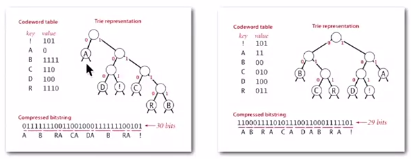
* Chars in leaves
* Codeword is path from root to leaf

Compression

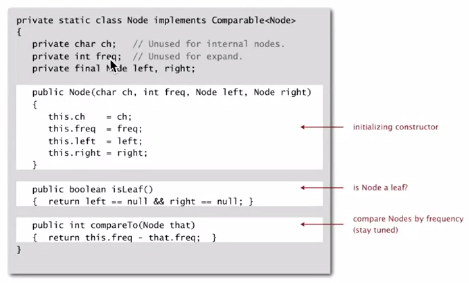
* Method 1: start at leaf; follow path up to the root; print bits in reverse
* Method 2: create ST of key-value pairs

Expansion

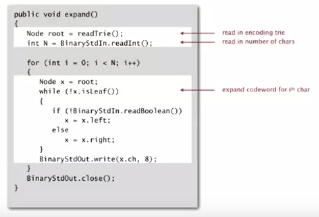
* Start at root
* Go left if bit is 0, go right if 1
* If leaf node, print char and return to root



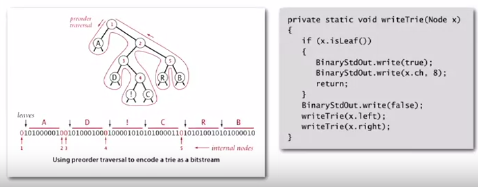
Trie node data type implementation in Java



Expansion implementation

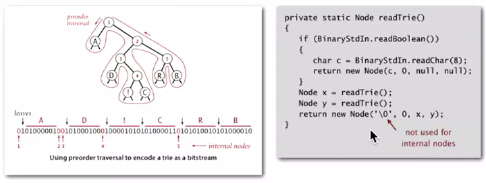


Writing the trie: write preorder traversal of trie; mark leaf and internal nodes with a bit



Note: if message is long, overhead of transmitting trie is small (all of possible characters in message).

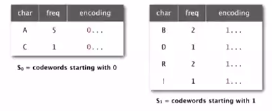
Read in the trie: reconstruct from preorder traversal of trie



Finding the best prefix-free code…

Shannon-Fado codes

* Partition symbols S into two subsets S0 and S1 of (roughly) equal freq
* Codewords for symbols in S0 start with 0; for symbols in S1 start with 1
* Recur in S0 and S1

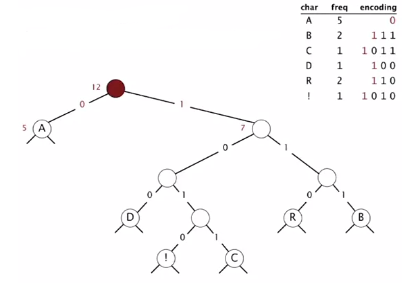


2 problems:

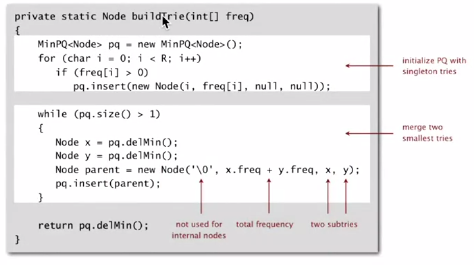
* How to divide the symbols?
* Not optimal

Huffman algorithm

1. Count frequency freq[i] for each char i in input
2. Start with one node corresponding to each char i (with weight freq[i])
3. Repeat until trie is formed:
   1. Select two tries with min weight freq[i] and freq[j]
   2. Merge into single trie with cumulative weight freq[i] + freq[j]



Used for JPEG, PDF, etc.

Java implementation

Proposition [Huffman 1950s] Huffman algorithm produces an optimal prefix-free code (no prefix-free code uses fewer bits than Huffman’s algorithm)  
 *Proof in textbook*

Implementation (2 passes)

* Pass 1: tabulate char frequencies and build trie
* Pass 2: encode file by traversing trie or lookup table

Running time: using a binary heap -> N (input size) + R log R (alphabet size)