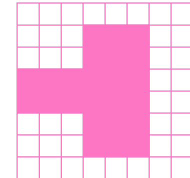
Data Compression

- Compression is a high-profile application
  - .zip, .mp3, .aac, .jpg, .gif, .gz, .mpg, …
  - What property of MP3 was a significant factor in what made P2P file sharing work?
- Why do we care?
  - Secondary storage capacity doubles every year
  - Disk space fills up quickly on every computer system
  - More data to compress than ever before
- Data compression facilitated by priority queue
  - All-time best assignment in a Compsci 100(e) course?
    - Subject to debate, of course

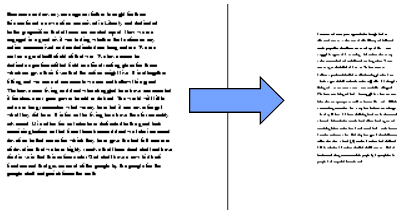
More on Compression

- What's the difference between compression techniques?
  - .mp3 files and .zip files?
  - .gif and .jpg?
  - Lossless and lossy
- Is it possible to compress (lossless) every file? Why?
- Lossy methods
  - Good for pictures, video, and audio (JPEG, MPEG, etc.)
- Lossless methods
  - Run-length encoding, Huffman, LZW, …



11 3 5 3 2 6 2 6 5 3 5 3 5 3 10

Text Compression



INPUT OUTPUT

- Input: String S
- Output: String S'
  - Shorter
  - S can be reconstructed from S'

Text Compression: Examples

Encodings

ASCII: 8 bits/character

Unicode: 16 bits/character

Symbol	ASCII	Fixed length	Var. length
a	01100001	000	000
b	01100010	001	11
c	01100011	010	01
d	01100100	011	001
e	01100101	100	10

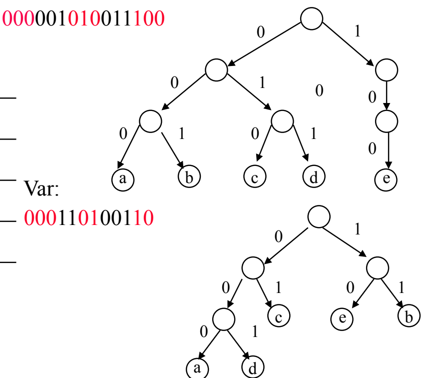
"abcde" in the different formats

ASCII:

01100001011000100110001101100100…

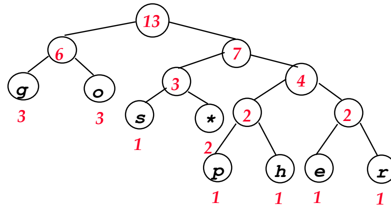
Fixed:

000001010011100



Huffman coding: *go go gophers*

	ASCII		3 bits	Huffman
g	103	*1100111*	000	00
o	111	*1101111*	001	01
p	112	*1110000*	010	1100
h	104	*1101000*	011	1101
e	101	*1100101*	100	1110
r	114	*1110010*	101	1111
s	115	*1110011*	110	101
sp.	32	*1000000*	111	101



- Encoding uses tree:
  - 0 left/1 right
  - How many bits? 37!!
  - Savings? Worth it?

Huffman Coding

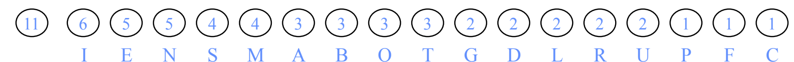
- D.A Huffman in early 1950's
- Before compressing data, analyze the input stream
- Represent data using variable length codes
- Variable length codes though *Prefix* codes
  - Each letter is assigned a codeword
  - Codeword is for a given letter is produced by traversing the Huffman tree
  - **Property**: *No codeword produced is the prefix of another*
  - Letters appearing frequently have short codewords, while those that appear rarely have longer ones
- Huffman coding is optimal *per-character* coding method

Building a Huffman tree

- Begin with a forest of single-node trees (leaves)
  - Each node/tree/leaf is weighted with character count
  - Node stores two values: character and count
  - There are *n* nodes in forest, *n* is size of alphabet?
- Repeat until there is only one node left: root of tree
  - Remove two minimally weighted trees from forest
  - Create new tree with minimal trees as children,
    - New tree root's weight: sum of children (character ignored)
- Does this process terminate? How do we get minimal trees?
  - Remove minimal trees, hummm……

Building a tree

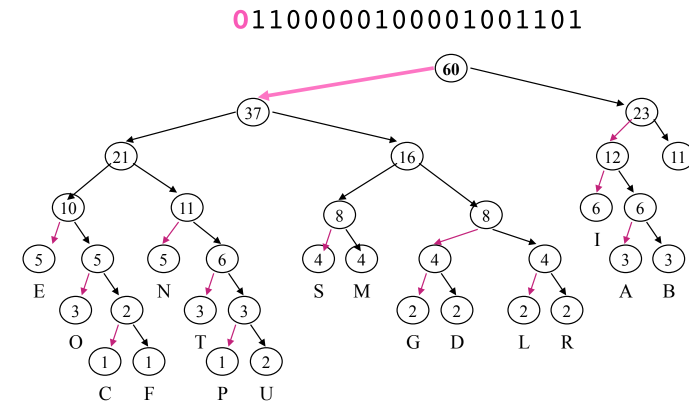
"A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS"



Properties of Huffman coding

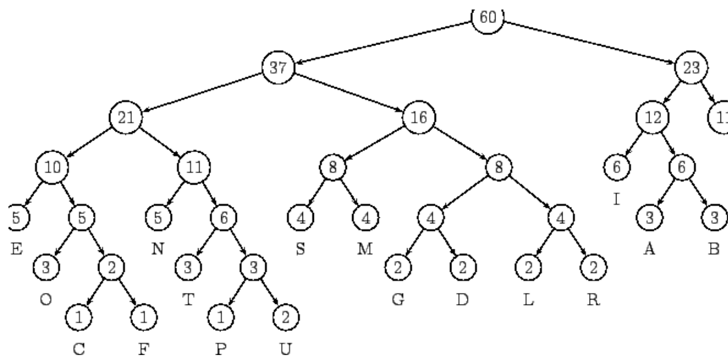
- Want to minimize weighted path length $L(T)$of tree T
- $$L(T) = \sum_{i \in Leaf(T)} d_i w_i$$
  - w_i is the weight or count of each codeword i
  - d_i is the leaf corresponding to codeword i
- How do we calculate character (codeword) frequencies?
- Huffman coding creates pretty full bushy trees?
  - When would it produce a "bad" tree?
- How do we produce coded compressed data from input efficiently?

Decoding a message



Huffman Tree 2

- "A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS"
  - E.g. " A SIMPLE" ⇔ "10101101001000101001110011100000"



Building a Huffman tree

- Begin with a forest of single-node trees/tries (leaves)
  - Each node/tree/leaf is weighted with character count
  - Node stores two values: character and count
- Repeat until there is only one node left: root of tree
  - Remove two minimally weighted trees from forest
  - Create new tree/internal node with minimal trees as children,
    - Weight is sum of children's weight (no char)
- How does process terminate? Finding minimum?
  - Remove minimal trees, hummm……

How do we create Huffman Tree/Trie?

- Insert weighted values into priority queue
  - What are initial weights? Why?
  -
- Remove minimal nodes, weight by sums, re-insert
  - Total number of nodes?

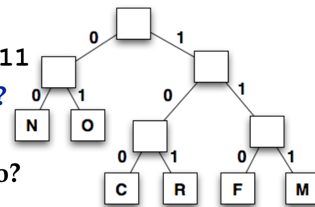
```
PriorityQueue<TreeNode> pq = new PriorityQueue<TreeNode>();
for(int k=0; k < freq.length; k++){
      pq.add(new TreeNode(k,freq[k],null,null));
}
while (pq.size() > 1){
   TreeNode left = pq.remove();
   TreeNode right = pq.remove();
   pq.add(new TreeNode(0,left.weight+right.weight,
                    left,right));
}
TreeNode root = pq.remove();
```


Creating compressed file

- Once we have new encodings, read every character
  - Write encoding, not the character, to compressed file
  - Why does this save bits?
  - What other information needed in compressed file?
- How do we uncompress?
  - How do we know foo.hf represents compressed file?
  - Is suffix sufficient? Alternatives?
- Why is Huffman coding a two-pass method?
  - Alternatives?

Uncompression with Huffman

- We need the trie to uncompress
  - 000100100010011001101111
- As we read a bit, what do we do?
  - Go left on 0, go right on 1
  - When do we stop? What to do?



- How do we get the trie?
  - How did we get it originally? Store 256 int/counts
    - How do we read counts?
  - How do we store a trie? 20 Questions relevance?
    - Reading a trie? Leaf indicator? Node values?

Other Huffman Issues

- What do we need to decode?
  - How did we encode? How will we decode?
  - What information needed for decoding?
- Reading and writing bits: chunks and stopping
  - Can you write 3 bits? Why not? Why?
  - PSEUDO_EOF
  - BitInputStream and BitOutputStream: API
- What should happen when the file won't compress?
  - Silently compress bigger? Warn user? Alternatives?

Huffman Complexities

- How do we measure? Size of input file, size of alphabet
  - Which is typically bigger?
- Accumulating character counts: ______
  - How can we do this in O(1) time, though not really
- Building the heap/priority queue from counts ____
  - Initializing heap guaranteed
- Building Huffman tree ____
  - Why?
- Create table of encodings from tree ____
  - Why?
- Write tree and compressed file _____

Good Compsci 100(e) Assignment?

- Array of character/chunk counts, or is this a map?
  - Map character/chunk to count, why array?
- Priority Queue for generating tree/trie
  - Do we need a heap implementation? Why?
- Tree traversals for code generation, uncompression
  - One recursive, one not, why and which?
- Deal with bits and chunks rather than ints and chars
  - The good, the bad, the ugly
- Create a working compression program
  - How would we deploy it? Make it better?
- Benchmark for analysis
  - What's a *corpus*?

Other methods

- Adaptive Huffman coding
- Lempel-Ziv algorithms
  - Build the coding table on the fly while reading document
  - Coding table changes dynamically
  - Protocol between encoder and decoder so that everyone is always using the right coding scheme
  - Works well in practice (`compress`, `gzip`, etc.)
- More complicated methods
  - Burrows-Wheeler (`bunzip2`)
  - PPM statistical methods

Data Compression

Year	Scheme	Bit/ Char
1967	ASCII	7.00
1950	Huffman	4.70
1977	Lempel-Ziv (LZ77)	3.94
1984	Lempel-Ziv-Welch (LZW) – Unix compress	3.32
1987	(LZH) used by zip and unzip	3.30
1987	Move-to-front	3.24
1987	gzip	2.71
1995	Burrows-Wheeler	2.29
1997	BOA (statistical data compression)	1.99

- Why is data compression important?
- How well can you compress files losslessly?
  - Is there a limit?
  - How to compare?
- How do you measure how much information?