Outline

PA1 Past Due

Last time:

- Binary Search Tree (BST)
- Binary Space Partition (BSP) Tree
- MinHeap and MaxHeap
- Priority queue

Today:

- Trie
- RLE and Huffman encoding
**Trie**

**trie**: from *retrieval*, rhymes with *try*, to differentiate from *tree*

A trie: a tree that uses parts of the key, as opposed to the whole key, to perform search. Whereas a tree associates keys with nodes, a trie associates keys with edges.

Example: for set of strings \{on, owe, owl, tip, to\}

Note: the handout’s **external node** is our leaf node, not our external (null) node.
Trie Definition

Let $S$ be a set of strings from alphabet $\Sigma$ such that no string in $S$ is a prefix of another

A *trie* for $S$ is an *ordered* tree $T$ such that:

- each edge of $T$ is labeled with symbols from $\Sigma$
- the ordering of edges of a node follows the canonical ordering of $\Sigma$
- labels of edges on the path from the root to any node in $T$ forms a prefix of a string in $S$

An $n$-ary trie is usually implemented with labels for the edges stored either at the children nodes or at the parent node.
Text Compression

Lossless (vs. lossy) compression

- run-length encoding
- statistical modelling: Huffman coding, arithmetic coding (not covered)
- dictionary-based encoding: LZ77, LZSS, LZ78, LZW

Readings:

- Dictionary method, LZ77, LZSS, LZ78
- LZW: http://www.dogma.net/markn/articles/lzw/lzw.htm
Run Length Encoding

A very simple encoding method:
for each repeated element, output the element and the count

Example: 1111111100000111100001111000011111110000
Output: 8150414041507140
Huffman Encoding

Example string: If a woodchuck could chuck wood!

ASCII encoding: 8 bits/char ⇒ requires 256 bits to encode (store in binary) the string

Observe: there are only 13 distinct symbols in example string, so 4 bits/char is sufficient to encode the string ⇒ requires 128 bits

Huffman encoding’s main ideas:
  • variable-length encoding: use different number of bits (code length) to represent different symbol
  • entropy encoding: assign smaller code to more frequently occurring symbol

Goal: \( \sum l(c) \cdot f(c) \) minimized, where \( c \) is each unique symbol in string, \( l(c) \) length of its code, and \( f(c) \) its frequency
**Huffman Encoding (contd)**

If a woodchuck could chuck wood!

Can be encoded using the following codes (for example):

<table>
<thead>
<tr>
<th>symbol</th>
<th>freq. ( f(c) )</th>
<th>code ( C(c) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>l</td>
<td>1</td>
<td>11111</td>
</tr>
<tr>
<td>f</td>
<td>1</td>
<td>11100</td>
</tr>
<tr>
<td>a</td>
<td>1</td>
<td>110101</td>
</tr>
<tr>
<td>l</td>
<td>1</td>
<td>11000</td>
</tr>
<tr>
<td>!</td>
<td>1</td>
<td>1101</td>
</tr>
<tr>
<td>w</td>
<td>2</td>
<td>1100</td>
</tr>
<tr>
<td>d</td>
<td>3</td>
<td>101</td>
</tr>
<tr>
<td>u</td>
<td>3</td>
<td>100</td>
</tr>
<tr>
<td>h</td>
<td>2</td>
<td>0111</td>
</tr>
<tr>
<td>k</td>
<td>2</td>
<td>0110</td>
</tr>
<tr>
<td>o</td>
<td>5</td>
<td>010</td>
</tr>
<tr>
<td>c</td>
<td>5</td>
<td>001</td>
</tr>
<tr>
<td>' '</td>
<td>5</td>
<td>000</td>
</tr>
</tbody>
</table>

Takes only 111 bits to encode the string:
Huffman Encoding

Code table construction:

- want: more frequently occurring symbols to have shorter codes
- given that it is a variable length code, so that there’ll be no confusion when decoding, no code can be a prefix of another
  (hence known as a prefix-free code, or simply, prefix code)

Huffman code decoding:

- need the code table (cost amortized over message)
- variable length code, but no code is a prefix of another!
Huffman Trie

Better known as Huffman tree

Huffman tree construction algorithm:

- count the frequency of occurrence of each symbol
- make each symbol a leaf node, with its frequency as its weight
- repeatedly combine trees with smallest weight first (break ties arbitrarily)
- weight of combine tree is the sum of its two children’s
- encode each symbol as the path from the root, with a left represented as 0, right 1

Example:
Huffman Tree Construction (contd)

Characteristics of Huffman trees:

- higher frequency symbols at shallower depth
- since all symbols are leaf nodes, no code is a prefix of another

Construct Huffman tree from the $|\Sigma|$ elements ($|\Sigma|$: alphabet size):

- implement as a MinHeap, where the “key” is the frequency of occurrence of each element of $\Sigma$
- take the two smallest elements off MinHeap, $O(\ )$
- make a tree of them, with the key of the new root node being the sum of the keys of the two children, $O(\ )$
- put new tree back into MinHeap, $O(\ )$

Total construction time: $O(\ )$
Encoding Time Complexity

Running times, \( n \) string length, \( |\Sigma| \) alphabet size

- frequency count: \( O(n) \)
- Huffman tree construction: \( O(|\Sigma| \log |\Sigma|) \)
- Total time: \( O(n + |\Sigma| \log |\Sigma|) \)

For binary data, treat each byte as a “character”
Compressing the Huffman Code Table

The Huffman code for any particular text is not unique. For example, the following are all acceptable:

<table>
<thead>
<tr>
<th>symbol</th>
<th>freq.</th>
<th>code</th>
<th>$C(c)$</th>
<th>$C'(c)$</th>
<th>$C''(c)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>,</td>
<td>5</td>
<td>000</td>
<td>001</td>
<td>000</td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>5</td>
<td>001</td>
<td>010</td>
<td>001</td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>3</td>
<td>101</td>
<td>100</td>
<td>010</td>
<td></td>
</tr>
<tr>
<td>o</td>
<td>5</td>
<td>010</td>
<td>000</td>
<td>011</td>
<td></td>
</tr>
<tr>
<td>u</td>
<td>3</td>
<td>100</td>
<td>101</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>2</td>
<td>1101</td>
<td>0110</td>
<td>1010</td>
<td></td>
</tr>
<tr>
<td>h</td>
<td>2</td>
<td>0111</td>
<td>1101</td>
<td>1011</td>
<td></td>
</tr>
<tr>
<td>k</td>
<td>2</td>
<td>0110</td>
<td>1110</td>
<td>1100</td>
<td></td>
</tr>
<tr>
<td>w</td>
<td>1</td>
<td>1100</td>
<td>0111</td>
<td>1101</td>
<td></td>
</tr>
<tr>
<td>a</td>
<td>1</td>
<td>11101</td>
<td>1100</td>
<td>1110</td>
<td></td>
</tr>
<tr>
<td>f</td>
<td>1</td>
<td>11110</td>
<td>1101</td>
<td>1110</td>
<td></td>
</tr>
<tr>
<td>l</td>
<td>1</td>
<td>11100</td>
<td>1111</td>
<td>1110</td>
<td></td>
</tr>
<tr>
<td>l</td>
<td>1</td>
<td>11111</td>
<td>11110</td>
<td>11111</td>
<td></td>
</tr>
</tbody>
</table>

The last column can be compressed into: 3’ ’cdou4!hwk5af1I

Sugih Jamin (jamin@eecs.umich.edu)
Compressing the Huffman Code Table (contd)

The last column was not created from a Huffman tree directly; the Huffman tree is used only to determine the code length of each symbol, then:

1. order symbols by code length
2. set the first of the shortest code length to all 0s
3. add 1 to the code of each subsequent symbol
4. when transiting from code length $k$ to code length $k + 1$, add 1 to the last length-$k$ code and use that as the prefix for the first length $k + 1$ code
5. set the $k + 1$st bit to 0 and repeat from Step 3

Code tables that follow these rules will have only one encoding for each symbol and still be prefix-free

Sugih Jamin (jamin@eecs.umich.edu)
Huffman and Run Length Encoding

String: If a woodchuck could chuck wood!
Output: 1111111110000111...
RLE: 914031...

The output of Huffman encoding can be further encoded using run-length encoding, or vice versa:

String: aabcccbbaaabaabccccaaacaaaaa
RLE: 2a1b3c1b3a1b2a1b3c2a1c5a
Then create a Huffman tree out of 2a, 1b, 3c, 3a, 1c, 5a