From Sorting to Heaps to Compression

- **Data Compression**
  - video on demand/set top box
  - jpeg in browsers
  - gzip, pkzip, compress, zip, ... for files (stacker?)

- **Lossy compression, Lossless compression**

- **Huffman coding**
  - possible to implement, reasonably good
  - uses lots of things we’ve studied, trees, priority queues, vectors, ...
  - leads to more advanced techniques: Lempel-Ziv
Priority Queues

- As an abstract data type (ADT) supports
  - add/insert: put an element into the priority queue
  - getMin: find the minimal (priority) element
  - deleteMin: delete the minimal element
  - (possible to have maximal queue too)

- Implement with different structures:
  - sorted linked list, vector, binary search tree, heap

```
insert    getMin    deleteMin
linked-list
vector
search tree
balanced tree
```
Heap: a data structure for priority queues

- modeled on binary trees, but implemented with array/vector
  > supports Insert and DeleteMin in $O(\log n)$ worst-case time
  > supports FindMin in $O(1)$ time and Insert in $O(1)$ average-case time
- Consider the following sorting method, complexity?

```c
void HeapSort(Vector<string> & a, int numElts)
{
    PQueue<string> pq;
    for(int k=0; k < numElts; k++) pq.insert(a[k]);
    for(int k=0; k < numElts; k++) pq.deleteMin(a[k]);
}
```

- we’ll return to heap implementation to see how the performance guarantees are realized
Towards Compression

- Each ASCII character is represented by 8 bits, one byte
  - bit is a binary digit, byte is a binary term
  - compress text: use fewer bits for frequent characters (does this come free?)

- 256 character values, \(2^8 = 256\), how many bits for 7 characters? for 38 characters? for 125 characters?

**go go gophers:** 8 different characters

- ASCII: 13 x 8 = 104 bits
- 3 bit code: 13 x 3 = 39 bits
- compressed: ???

<table>
<thead>
<tr>
<th>ASCII</th>
<th>3 bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>103 1100111 000</td>
</tr>
<tr>
<td>o</td>
<td>111 1101111 001</td>
</tr>
<tr>
<td>p</td>
<td>112 1110000 010</td>
</tr>
<tr>
<td>h</td>
<td>104 1101000 011</td>
</tr>
<tr>
<td>e</td>
<td>101 1100101 100</td>
</tr>
<tr>
<td>r</td>
<td>114 1110010 101</td>
</tr>
<tr>
<td>s</td>
<td>115 1110011 110</td>
</tr>
<tr>
<td>sp.</td>
<td>32 1000000 111</td>
</tr>
</tbody>
</table>
Huffman coding: *go go gophers*

<table>
<thead>
<tr>
<th>ASCII</th>
<th>3 bits</th>
<th>Huffman</th>
</tr>
</thead>
<tbody>
<tr>
<td>g</td>
<td>103</td>
<td>1100111</td>
</tr>
<tr>
<td>o</td>
<td>111</td>
<td>1101111</td>
</tr>
<tr>
<td>p</td>
<td>112</td>
<td>1110000</td>
</tr>
<tr>
<td>h</td>
<td>104</td>
<td>1101000</td>
</tr>
<tr>
<td>e</td>
<td>101</td>
<td>1100101</td>
</tr>
<tr>
<td>r</td>
<td>114</td>
<td>1110010</td>
</tr>
<tr>
<td>s</td>
<td>115</td>
<td>1110011</td>
</tr>
<tr>
<td>sp.</td>
<td>32</td>
<td>1000000</td>
</tr>
</tbody>
</table>

- choose two fewest # occ’s
- combine nodes, add occ’s
- repeat

- How many bits?
Properties of Huffman code

- Prefix property, no code is prefix of another code
- Optimal per character compression
- Where do frequencies come from?
- Decode: need tree

1000111101001110100000110101111011110001

Duke CPS 100