Matt Elgart
Huffman Analysis

Benchmarking the code with the HuffMark class:
Both an implementation of a standard count format header and a tree format header were used. The benchmarking was done for each implementation as seen below.

SimpleHuffProcessor:

calgary:

```
total bytes read: 3257641
total compressed bytes 1847346
total percent compression 43.292
compression time: 3.334
```

waterloo:

```
total bytes read: 12472452
total compressed bytes 10207097
total percent compression 18.163
compression time: 17.902
```

TreeHuffProcessor:

calgary:

```
total bytes read: 3257641
total compressed bytes 1830946
total percent compression 43.795
compression time: 3.371
```

waterloo:

```
total bytes read: 12472452
total compressed bytes 10200597
total percent compression 18.215
compression time: 18.384
```

For both implementations, the results were fairly similar. In general, the tree format implementation took just slightly longer (the difference between the two formats was on the order of 1% of the total compression time), but also compressed just slightly more data (again, the difference was on the order of 1% of the total percent compression in favor of the tree implementation).

There was, however, a marked difference in times and percent compression between the two types of files being compressed: text files for calgary and images for waterloo. Percentage wise, the text files on average were compressed more than twice the image files were (~43% and ~18%, respectively). The image files also took far longer to compress, as seen in the data above.

The reason for the differences in compression percentages could lie in how the compression algorithm operates. The Huffman coding algorithm uses the frequency with which characters (or, more generally, chunks of bits) appear in a file in order to create encodings for each one. By using a PriorityQueue to create a tree to represent the characters/chunks, the algorithm is able to assign smaller encodings to characters/chunks that appear more frequently, and vice versa. This process means that the algorithm will be most efficient in situations where the original file has large imbalances in the frequencies of its various characters/chunks. The English language (and written language in general) happens to be a fairly good example of this scenario, where some characters (take, for example, the letter “e”) are used far more often than others (e.g. the letter “q”). We might therefore expect written text files to have more compression than a file with a comparatively more even distribution of bit-chunks. One can imagine that an image file might have more of this kind of even distribution, as bit-chunks can have large variation between given individual files. This difference might be the cause for which the text files compress so much more than the image files do when using the Huffman coding algorithm.

Also, as a file gets compressed, compressing it again will reduce the percent effectiveness of the process. Two possible reasons for this result are the necessity for a header, and the possibility that as a file gets compressed, the previously mentioned variance in frequencies gets evened out.