The ceiling operator ⌈ ⌉ is introduced because Huffman codes use an integer number of bits to encode each symbol.

We could get rid of the ceiling operation if we replaced the Huffman code with an Arithmetic code, but it is probably not worth the effort.

Lookup time is relatively slow. Technically, lookup time is *O*(*N*), because one has to start at the beginning and add up the interarrivals to reconstruct the hash codes. McIlroy actually introduced a small table on the side with hash codes and offsets so one could seek to these offsets and avoid starting at the beginning every time. Even so, our experiments will show that HashTBO is an order of magnitude slower than ZipTBO.

Accuracy is also an issue. Fortunately, we don't have a problem with dropouts. If a word is in the dictionary, we aren't going to misplace it. But two words in the dictionary could hash to the same value. In addition, a word that is not in the dictionary could hash to the same value as a word that is in the dictionary. For McIlroy's application (detecting spelling errors), the only concern is the last possibility. McIlroy did what he could do to mitigate false positive errors by increasing *P* as much as he could, subject to the memory constraint (the PDP-11 address space of 64k bytes).

We recommend these heroics when space dominates other concerns (time and accuracy).

5 Golomb Coding

Golomb coding takes advantage of the sparseness in the interarrivals between hash codes. Let's start with a simple recipe. Let *t* be an interarrival. We will decompose t into a pair of a quotient (t_q) and a remainder (t_r). That is, let $t = t_q m + t_r$ where $t_q = \lfloor t/m \rfloor$ and $t_r = t \bmod m$. We choose m to be a power of two near $m \approx \left\lceil \frac{E[t]}{2} \right\rceil = \left\lceil \frac{P}{2N} \right\rceil$, where $E[t]$ is the expected value of the interarrivals, defined below. Store t_q in unary and t_r in binary.

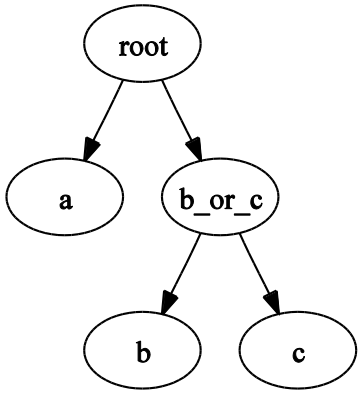
Binary codes are standard, but unary is not. To encode a number *z* in unary, simply write out a sequence of *z*-1 zeros followed by a 1. Thus, it takes *z* bits to encode the number z in unary, as opposed to $\log_2 z$ bits in binary.

This recipe consumes $t_q + \log_2 m$ bits. The first term is for the unary piece and the second term is for the binary piece.

Why does this recipe make sense? As mentioned above, a Golomb code is a Huffman code for an infinite alphabet with exponential probabilities. We illustrate Huffman codes for infinite alphabets by starting with a simple example of a small (very finite) alphabet with just three symbols: {*a*, *b*, *c*}. Assume that half of the time, we see *a*, and the rest of the time we see *b* or *c*, with equal probabilities:

Symbol	Code	Length	Pr
A	0	1	50%
B	10	2	25%
C	11	2	25%

The Huffman code in the table above can be read off the binary tree below. We write out a 0 whenever we take a left branch and a 1 whenever we take a right branch. The Huffman tree is constructed so that the two branches are equally likely (or at least as close as possible to equally likely).



Now, let's consider an infinite alphabet where $\Pr(a) = \frac{1}{2}$, $\Pr(b) = \frac{1}{4}$ and the probability of the t+1st symbol is $\Pr(t) = (1-\beta)\beta^t$ where $\beta = \frac{1}{2}$. In this case, we have the following code, which is simply *t* in unary. That is, we write out $t-1$ zeros followed by a 1.

Symbol	Code	Length	Pr
A	1	1	2^{-1}
B	01	2	2^{-2}
C	001	3	2^{-3}