CIS 121 — Data Structures and Algorithms
Huffman Coding
October 3, 2019

Adapted from Kleinberg and Tardos's *Algorithm Design*

From Text to Bits

Computers ultimately operate on sequences of bits. As a result, one needs encoding schemes that take text written in richer alphabets (such as the alphabets underpinning human languages) and converts this text into strings of bits. Moreover, in many cases, it is useful to do so in a way that compresses the data. For example, when large files need to be shipped across communication networks, or stored on hard disks, it is important to represent them as compactly as possible without losing information. As a result, compression algorithms are a big focus in research. Before we jump into data compression algorithms, though, let's return to the problem of turning characters into bit sequences.

The simplest way to do this would be to use a fixed number of bits for each symbol in the alphabet, and then just concatenate the bit strings for each symbol to form the text. Since you can form 2^b different sequences using b bits, you would need $\lceil \lg n \rceil$-bit symbols to encode an alphabet of n characters. For example, to encode the 26 letters of the English alphabet, plus "space," you'd need 5 bit symbols. In fact, encoding schemes like ASCII work precisely this way, except they use a larger number of bits per symbol (eight in the case of ASCII) in order to handle larger alphabets which include capital letters, parenthesis, and a bunch of other special symbols.

This method certainly gets the job done, but we should ask ourselves: is there anything more we could ask for from an encoding scheme, maybe with respect to data compression? We can't just use fewer bits per encoding: if we only used four bits in the example above, we could only have $2^4 = 16$ symbols. However, what if we used fewer bits for *some* characters? As it currently stands (using the above example), we use five bits for every character, so obviously the average number of bits per character is five. In some cases, though, we may be able to use fewer than five bits per character *on average*. Think about it: letters like e, t and a get used much more frequently than q and x (by more than an order of magnitude, in fact). So it's a big waste to translate them all into the same number of bits. Instead, we could use a small number of bits for the frequent letters, and a larger number of bits for the less frequent ones, and hope to end up using fewer than five bits per letter when we average over a long string of text.

Variable-Length Encoding Schemes

In fact, this idea was used by Samuel Morse when developing *Morse Code*, which translates letters to dots and dashes (you can think of these as 0's and 1's). Morse consulted local printing presses to get frequency estimates for the letters in English and constructed his namesake code accordingly ("e" is a single dot—0, "t" a single dash—1, "a" is dot-dash—01, etc.).

Morse code, however, uses such short strings for letters that the encoding of words becomes ambiguous. For example, the string 0101 could correspond to any one of $eta, aa, etet$, or aet. Morse dealt with this ambiguity by putting pauses between letters. While this certainly eliminates ambiguity, it also means

that now we require three symbols (dot, dash, pause) instead of only two (dot, dash). Since our goal is to translate into bits, we need to use a different method.

Prefix Codes

The ambiguity in Morse code arises because there are pairs of letters where the bit string that encodes one is a *prefix* of the bit string that encodes the other. To eliminate this problem, and hence to obtain an encoding scheme that has a well-defined interpretation for every sequence of bits, we need the encoding to be *prefix-free*:

Definition A prefix code for a set S of letters is a function γ that maps each letters $x \in S$ to some sequence of zeros and ones, in such a way that for any distinct $x, y \in S$, the sequence $\gamma(x)$ is not a prefix of the sequence $\gamma(y)$.

Why does this ensure every encoding has a well-defined interpretation? Suppose we have a text consisting of letters $x_1x_2...x_n$. If we encode each letter as a bit sequence using an encoding function γ, then concatenate all the bit sequences together to get $\gamma(x_1)\gamma(x_2)...\gamma(x_n)$, then reconstructing the original text can be done by following these steps:

- Scan the bit sequence from left to right
- As soon as you've seen enough bits to match the encoding of some letter, output this as the first letter of the text. This must be the correct first letter, since no shorter or longer prefix of the bit sequence could encode any other letter
- Now delete the corresponding set of bits from the front of the message and iterate.

As an example, suppose $S = \{a, b, c, d, e\}$ and we have an encoding γ specified by

$$\gamma(a) = 11$$
$$\gamma(b) = 01$$
$$\gamma(c) = 001$$
$$\gamma(d) = 10$$
$$\gamma(e) = 000$$

Given the string 0010000011101, we see that c must be the first character, so now we are left with 0000011101. e is clearly the second character, leaving 0011101. c is next, leaving 1101. Hence a followed by b end the string. The final translation is *cecab*. The fact that γ is prefix free makes the translation straightforward.

Optimal Prefix Codes

We ultimately want to give more frequent characters shorter encodings. First we must introduce some notation.

Assume S is our alphabet. For each character $x \in S$, we assume there is a frequency f_x representing the fraction of letters in the text equal to x. That is, we assume a probability distribution over the letters in

the texts we will be encoding. In a text with n letters, we expect nf_x of them to be equal to x. Also, we must have

$$\sum_{x \in S} f_x = 1$$

since the frequencies must sum to 1.

If we use a prefix code γ to encode a text of length n, what is the total length of the encoding? Writing $|\gamma(x)|$ to denote the number of bits in the encoding $\gamma(x)$, the expected encoding length of the full text is given by

$$\text{encoding length} = \sum_{x \in S} nf_x \cdot |\gamma(x)| = n \sum_{x \in S} f_x \cdot |\gamma(x)|$$

That is, we simply take the sum over all letters $x \in S$ of the number of times x occurs in the text (nf_x) times the length of the bit string $\gamma(x)$ used to encode x. The average number of bits per letter is simply the encoding length divided by the number of letters:

$$\text{Average bits per letter} = \text{ABL}(\gamma) = \sum_{x \in S} f_x \cdot |\gamma(x)|$$

Using the previous example where $S = \{a, b, c, d, e\}$, suppose the frequencies were given by

$$f_a = 0.32 \quad f_b = 0.25 \quad f_c = 0.20 \quad f_d = 0.18 \quad f_e = 0.05$$

Then using the same encoding from above, we have

$$\text{ABL}(\gamma) = 0.32 \cdot 2 + 0.25 \cdot 2 + 0.20 \cdot 3 + 0.18 \cdot 2 + 0.05 \cdot 3 = 2.25$$

Note that a fixed length encoding with five characters would require $\lceil \lg 5 \rceil = 3$ bits per character. Thus using the code γ reduces the bits per letter from 3 to 2.25, a saving of 25%. In fact, there are even better encodings for this example.

Definition An optimal prefix code is as efficient as possible. That is, given an alphabet S and a set of frequencies for the letters in S, it minimizes the average number of bits per letter $\text{ABL}(\gamma) = \sum_{x \in S} f_x \cdot |\gamma(x)|$

This leads us to the underlying question: Given an alphabet S and a set of frequencies for the letters in S, how do we produce an optimal prefix code?

Huffman Encoding

Binary Trees and Prefix Codes

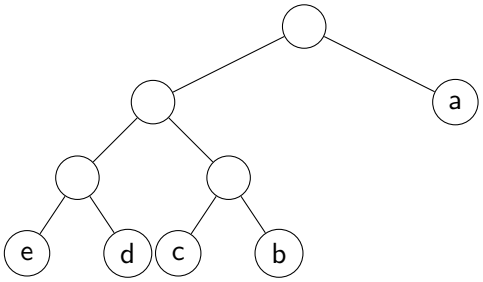
A key insight into this problem is developing a tree-based representation of prefix codes. Suppose we take a rooted tree T in which each node that is not a leaf has exactly two children; i.e. T is a *binary tree*. Further, suppose that the number of leaves is equal to the size of the alphabet S, and we label each leaf with a distinct letter in S.

Such a labeled binary tree T naturally describes a prefix code. For each letter $x \in S$, we follow the path from the root to the leaf labeled x. Each time the path goes from a node to its left child, we write down a 0, and each time the path goes from a node to its right child, we write down a 1. We take the resulting string of bits as the encoding for x.

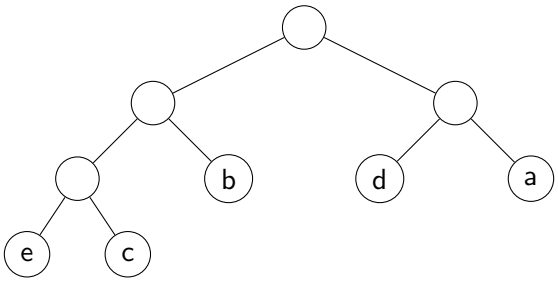
Lemma 1. *The encoding of S constructed from T as described above is a prefix code.*

Proof. In order for the encoding of x to be a prefix of the encoding for some other character y, the path from the root to x would have to be a prefix of the path from the root to y. But this is the same as saying that x would be on the path from the root to y, which isn't possible since x is a leaf. □

In fact, this relationship between binary trees and prefix codes goes the other way too: given a prefix code γ, we can build a binary tree recursively as follows. We start with a root. All letters $x \in S$ whose encodings begin with a 0 will be leaves in the left subtree of the root, and all the letters $y \in S$ whose encodings begin with a 1 will be leaves in the right subtree of the root. Then, just build the two subtrees recursively using this rule.



Character	Encoding
a	1
b	011
c	010
d	001
e	000



Character	Encoding
a	11
b	01
c	001
d	10
e	000

Figure 1: Examples of the correspondence between prefix codes and binary trees.

Thus by this equivalence, the search for an optimal prefix code can be viewed as the search for a binary tree T, together with a labeling of leaves of T, that minimizes the average number of bits per letter. Moreover, this average quantity has a natural interpretation in terms of the structure of T: the length of the encoding of a letter $x \in S$ is simply the length of the path from the root to the leaf labeled with x, a quantity which has a special name:

Definition The depth of a leaf in a binary tree is the length of the path from the root to that leaf.

The depth of a leaf v in tree T is denoted $\text{depth}_T(v)$.

Thus we seek the labeled tree that minimizes the weighted average of the depths of all leaves, where the average is weighted by the frequencies of the letters that label the leaves. We use $\text{ABL}(T)$ to denote this quantity. If γ is the prefix code corresponding to T, then

$$\text{ABL}(T) = \sum_{x \in S} f_x \cdot \text{depth}_T(x) = \sum_{x \in S} f_x \cdot |\gamma(x)| = \text{ABL}(\gamma)$$

One thing to note immediately about the optimal binary tree T is that it is *full*. That is, each node that is not a leaf has exactly two children. To see why, suppose the optimal tree weren't full. Then we could take any node with only one child, remove it from the tree, and replace it by its lone child. This would create a new tree T' such that $\text{ABL}(T') < \text{ABL}(T)$, contradicting the optimality of T. We record this fact below:

Lemma 2. *The binary tree corresponding to an optimal prefix code is full.*

A First Attempt: The Top-Down Approach

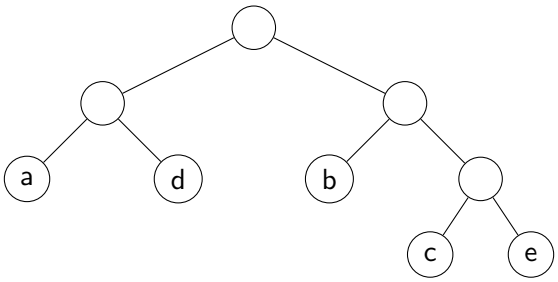
Intuitively, our goal is to produce a labeled binary tree in which the leaves are as close to the root as possible. This will give us a small average leaf depth.

A natural way to do this would be to try building a tree from the top down by "packing" the leaves as tightly as possible, perhaps using a divide-and-conquer method. So suppose we try to split the alphabet S into two sets, S_1 and S_2 such that the total frequency of the letters in each set is as close to $1/2$ as possible. We then recursively construct prefix codes for S_1 and S_2 independently, then make these the two subtrees of the root.

In fact, this type of encoding scheme is called the *Shannon-Fano* code, named after two major figures in information theory, Claude Shannon and Robert Fano. Shannon-Fano codes work OK in practice, but unfortunately it doesn't lead to an optimal prefix code. For example, consider our five letter alphabet $S = \{a, b, c, d, e\}$ with frequencies

$$f_a = 0.32 \quad f_b = 0.25 \quad f_c = 0.20 \quad f_d = 0.18 \quad f_e = 0.05$$

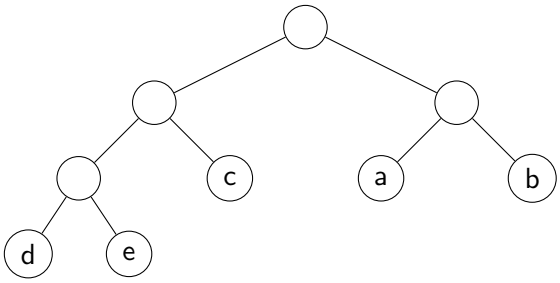
There is a unique way to split the alphabet into two sets of equal frequency: $\{a, d\}$ and $\{b, c, e\}$. Recursing on $\{a, d\}$, we can use a single bit to encode each. Recursing on $\{b, c, e\}$, we need to go one more level deeper in the recursion, and again, there is a unique way to divide the set into two subsets of equal frequency. The resulting code corresponds to:



Character	Encoding
a	00
b	10
c	110
d	01
e	111

Figure 2: A non-optimal prefix code for the given alphabet S.

However, it turns out this is NOT optimal. In fact, the optimal code is given by:



Character	Encoding
a	10
b	11
c	01
d	000
e	001

Figure 3: An optimal prefix code for the given alphabet S.

You can verify this is better than the Shannon-Fano tree by computing the value of $\text{ABL}(T)$ for each of the above trees. A quick way to see that the tree in Figure 3 is better than the Shannon-Fano is to notice that the only characters whose depth differ between the trees are c and d. Since d has a lower frequency than c, it should have greater depth.

David Huffman, a graduate student who had taken a class by Fano, decided to take up the problem for himself. This would eventually lead to Huffman's algorithm and Huffman coding, which always produce optimal prefix-free codes.

Huffman's Algorithm

In searching for an efficient algorithm or solving a tough problem, it often helps to assume—as a thought experiment—that you know something about the optimal solution, and then to see how you would make use of this partial knowledge in finding the complete solution. Applying this technique to the current problem, we ask: What if we knew the binary tree T^* that corresponded to the optimal prefix code, but not the labeling of the leaves? To complete the solution, we would need to figure out which letter should label which leaf of T^*, and then we'd have our code.

It turns out, this is rather simple to do.

Proposition 1. *Suppose that u and v are leaves of T^*, such that $\text{depth}(u) < \text{depth}(v)$. Further, suppose that in a labeling of T^* corresponding to an optimal prefix code, leaf u is labeled with character $y \in S$ and leaf v is labeled with $z \in S$. Then $f_y \geq f_z$.*

Proof. We will use a common technique called an *exchange argument* to prove this. Seeking contradiction, suppose that $f_y < f_z$. Consider the code obtained by exchanging the labels at the nodes u and v. In the expression for the average number of bits per letter, $\text{ABL}(T^*) = \sum_{x \in S} f_x \cdot \text{depth}(x)$, the effect of the exchange is as follows: the multiplier on f_y increases from $\text{depth}(u)$ to $\text{depth}(v)$, and the multiplier on f_z decreases by the same amount. All other terms remain the same.

Thus the change to the overall sum is $(\text{depth}(v) - \text{depth}(u))(f_y - f_z)$. If $f_y < f_z$, this change is a negative number, contradicting the supposed optimality of the prefix code that we had before the exchange. □

For example, as state above, it is easy to see that the tree in Figure 2 is not optimal, since we can exchange

d and c to achieve the more optimal code in Figure 3.

This proposition tells us how to label the tree T^* should someone give it to us: We go through the leaves in order of increasing depth, assigning letters in order of decreasing frequency. Note that, among labels assigned to leaves at the same depth, *it doesn't matter which label we assign to which leaf.* Since the depths are all the same, the corresponding multipliers in the expression $\text{ABL}(T^*) = \sum_{x \in S} f_x \cdot |\gamma(x)|$ are all the same too.

Now we have reduced the problem to finding the optimal tree T^*. How do we do this? It helps to think about the labelling process from the bottom-up, i.e. by considering the leaves at the greatest depths (which thus receive the labels with the lowest frequencies). Specifically, let v be a leaf in T^* whose depth is a large as possible. Leaf v has a parent u, and since T^* must be full, u has another child, w, which is a *sibling* of v. Note that w is also a leaf in T^*, otherwise, v would not be the deepest possible leaf.

Now, since v and w are siblings at the greatest depth, our level-by-level labelling process will reach them last. The leaves at this deepest level will receive the lowest frequency letters, and since we argued above that the order in which we assign these letters to the leaves within this level doesn't matter, there is an optimal labelling in which v and w get the two lowest-frequency letters over all:

Lemma 3. *There is an optimal prefix code, corresponding to tree T^*, in which the two lowest-frequency letters are assigned to leaves that are siblings in T^*.*

Designing the Algorithm

Suppose that y and z are the two lowest frequency letters in S (ties can be broken arbitrarily). Lemma 3 tells us that it is safe to assume that y and z are siblings in an optimal prefix tree. In fact, this directly suggests an algorithm: we replace y and z with a meta-letter w, whose frequency is $f_y + f_z$. That is, we link y and z together in the prefix tree, making them siblings with a common parent w. This step makes the alphabet one letter smaller, so we can recursively find a prefix code for the smaller alphabet. Once we reach an alphabet with one letter, we are done: this one letter is the root of our tree, and we can "unravel" or "open up" the process to obtain a prefix code for the original alphabet S.

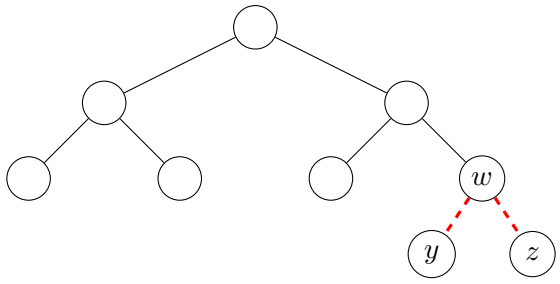


Figure 4: We take the two lowest frequency letters y and z, combine them to form a meta-letter w, and recurse.

Huffman's Algorithm

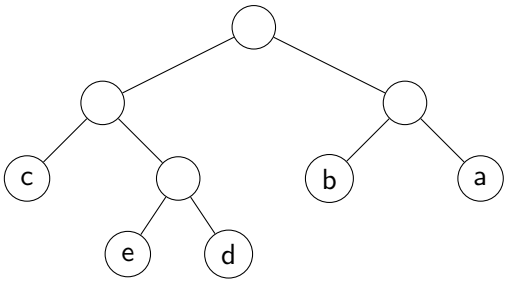
Input: A set of characters S, together with the frequencies of each character.
Output: An optimal, prefix-free code for S.

```
Huffman(S)
    If |S| = 2 then
        Encode one letter using 0 and the other letter using 1
    Else
        Let y,z be the two lowest-frequency letters

        Form a new alphabet S' by deleting y and z and
            replacing them with a new letter w of frequency f_w

        Recursively construct a prefix code for S' with tree T'

        Define a prefix code for S as follows:
            Start with T'
            Take the leaf labeled w and add y and z as its
                two children
```

Before jumping into the analysis of the algorithm, let's trace out how it would work on our running example: $S = \{a, b, c, d, e\}$ with frequencies

$$f_a = 0.32 \quad f_b = 0.25 \quad f_c = 0.20 \quad f_d = 0.18 \quad f_e = 0.05$$

We'd first merge d and e into a single letter, denoted (ed), of frequency $0.18+0.05 = 0.23$. We now recurse on the alphabet $S' = \{a, b, c, (ed)\}$. The two lowest frequency letters are now c and (ed), so we merge these into the single letter $(c(ed))$ of frequency $0.20 + 0.23 = 0.43$. Next we merge a and b to get the alphabet $\{(ba), (c(ed))\}$. Lastly, we perform one final merge to get the single letter $((c(ed))(ba))$, and we are done. Here, the parenthesis structure is meant to represent the structure of the corresponding prefix tree, which looks like:



Note the contrast with the Shannon-Fano codes. Instead of a divide-and-conquer, top-down approach, Huffman's algorithm uses a bottom-up, "greedy" approach. Here, "greedy" just means that at each step, we make local, short-sighted decision (merge the two lowest frequency nodes without worrying about how they will fit into the overall structure) that leads to a globally optimal outcome (an optimal prefix code for the whole alphabet).

Correctness of Huffman's Algorithm

Theorem 1. *The Huffman code for a given alphabet S achieves the minimum average number of bits per letter of any prefix code.*

Proof. Since the algorithm has a recursive structure, it is natural to try to prove optimality by induction on the size of the alphabet (CLRS provides a non-inductive proof). Clearly it is optimal for one and two-letter alphabets (it uses only one bit per letter). So suppose inductively that it is optimal for all alphabets of size $k-1$, where $k-1 \geq 2$ and consider an input consisting of an alphabet of size k.

Now, let y and z be the lowest frequency letters in S (ties broken arbitrarily). The algorithm proceeds by merging y and z into a single meta-letter w with $f_w = f_y + f_z$, and forming a new alphabet $S' = (S \cup \{w\}) - \{y, z\}$. The algorithm then recurses on S'. By the induction hypothesis, this recursive call produces an optimal prefix code for S', represented by a labeled binary tree T'. It then extends this into a tree T for S by attaching the leaves labeled y and z as the children of the node in T' labelled w.

We have the following relationship:

Lemma 4. *With T, T', and w defined as above, we have:*

$$ABL(T') = ABL(T) - f_w$$

Proof. The depth of each letter in S other than y and z is the same in both T and T'. Also, the depths of y and z in T are each one greater than the depth of w in T'. Using this, plus the fact that $f_w = f_y + f_z$, we have

$$\begin{aligned}
\mathrm{ABL}(T) &= \sum_{x \in S} f_x \cdot \mathrm{depth}_T(x) \\
&= f_y \cdot \mathrm{depth}_T(y) + f_z \cdot \mathrm{depth}_T(z) + \sum_{x \neq y,z} f_x \cdot \mathrm{depth}_T(x) \\
&= (f_y + f_z)(1 + \mathrm{depth}_{T'}(w)) + \sum_{x \neq y,z} f_x \cdot \mathrm{depth}_{T'}(x) \\
&= f_w + f_w \cdot \mathrm{depth}_{T'}(w) + \sum_{x \neq y,z} f_x \cdot \mathrm{depth}_{T'}(x) \\
&= f_w + \sum_{x \in S'} f_x \cdot \mathrm{depth}_{T'}(x) \\
&= f_w + \mathrm{ABL}(T')
\end{aligned}$$

□

Now we can prove optimality as follows: Seeking contradiction, suppose the tree T produced by Huffman's algorithm is not optimal. Then there exists some labeled binary tree Z such that $\mathrm{ABL}(Z) < \mathrm{ABL}(T)$. By **Lemma 3**, we can WLOG assume Z has y and z as siblings.

If we delete the leaves labeled y and z from Z, and label their former parent with w, we get a tree Z' that defines a prefix code for S'. In the same way that T is obtained from T', the tree Z is obtained

from Z' by adding leaves from y and z below w. Thus **Lemma 4** applies to Z and Z' as well, so $\text{ABL}(Z') = \text{ABL}(Z) - f_w$

But we have assumed that $\text{ABL}(Z) < \text{ABL}(T)$. Subtracting f_w from both sides of this inequality tells us that $\text{ABL}(Z') < \text{ABL}(T')$, contradicting the optimality of T' as a prefix code for S' (which was given by the induction hypothesis). □

Running Time

Let n be the number of characters in the alphabet S. The recursive calls of the algorithm define a sequence of $n-1$ iterations over smaller and smaller alphabets until reaching the base case. Identifying the lowest frequency letters can be done in a single scan of the alphabet in $O(n)$ time, and so summing over the $n-1$ iterations, the runtime is $T(n) \approx n + (n-1) + ... + 2 + 1 = \Theta(n^2)$

However, note that the algorithm requires finding the minimum of a set of elements. Thus we should expect to use a data structure that makes it easy to find the minimum of a set of elements quickly. A priority queue works well here. In this case, the keys are the frequencies. In each iteration, we just extract the minimum twice (to get the two lowest frequency letters), and then insert a new letter whose key is the sum of these two minimum frequencies, building the tree as we go. Our priority queue then contains a representation of the alphabet needed for the next iteration.

Using a binary-heap implementation of a priority queue, we know that each insertion and extract-min operation take $O(\log n)$ time. Hence summing over all n iterations gives a total running time of $O(n \log n)$.

This leads to a cleaner formulation of Huffman's algorithm:

Huffman's Algorithm

Input: A set of characters S, together with the frequencies of each character.
Output: An optimal, prefix-free code for S.

```
Huffman(S)
    Let Q be a priority queue containing all elements in S
        with frequencies as keys

    for i = 1 to n do
        allocate a new node z
        x = ExtractMin(Q)
        y = ExtractMin(Q)
        z.left = x
        z.right = y
        f_z = f_x + f_y
        Insert(Q, z)

    return ExtractMin(Q) // the root of the tree
```

Extensions

While Huffman works well in many cases, it by no means is the solution for all data compression problems. For example, consider transmitting black-and-white images, where each image is a 1000-by-1000 array of black or white pixels. Further, suppose a typical image is almost entirely white: only about 1000 of the million pixels are black. If we wanted to compress this image using prefix codes, it wouldn't work so well: we'd have a text of length one million over the two letter alphabet {black, white}. As a result, the text is already encoded using one-bit per letter, so no significant compression is really possible.

Intuitively, though, such images should be highly compressible. One possible compression scheme is to represent each image by a set of (x, y) coordinates denoting which pixels are black.

Another drawback of prefix codes is that they cannot *adapt* to changes in text. For example, suppose we are trying to encode the output of a program that produces a long sequence of letters from the set $\{a, b, c, d\}$. Further suppose that for the first half of this sequence, the letters a and b occur equally frequently, while c and d do not occur at all. Huffman's algorithm would view the entire text as one where each letter is equally frequent (all letters occur 1/4 of the time), and thus assign two bits per letter.

But what's really happening here is that the frequency remains stable for half the text, and then it changes radically. So one could get away with just one bit per letter plus a bit of extra overhead. Consider, for example, the following scheme:

- Begin with an encoding in which a is represented by 0 and b is represented by 1
- Halfway into the sequence, insert an instruction that says "Change the encoding to 0 represents c and 1 represents d" (this could be some kind of reserved special character).
- Use the new encoding for the rest of the sequence.

The point is that investing a small amount of space to describe a new encoding can pay off many times over if it reduces the average number of bits per letter over a long run of text that follows. Such approaches, which change the encoding in midstream, are called *adaptive* compression schemes, and for many kinds of data they lead to significant improvements over the static Huffman method.