This lecture talks about the format of a compressed file and how to reconstruct the code tree based on the beginning of the file.

As we know from the previous lecture, Huffman coding is based on the occurrences of the characters in the data. Thus, the code depends on the data. Two different files may contain different characters. Even if two input files have the same characters, the occurrences of the same character may be different. Thus, the code tree is specific to each data file.

We can use different formats for the files and different ways to describe the code trees, as long as the compression programs and the decompression programs agree. In this class, we will use this file format:

A compressed file has three parts. At the top is the code tree. The second part is the length of the original file. The first two parts are the header of the file. The third part is the actual data that uses the compression code.

We need a way to describe the code tree. The description must be unique for each tree. There are different ways describing a binary tree. In homework 15, you have learned how to build a binary tree from in-order and post-order traversals.

This lecture explains another way to describe a binary tree for the code tree. Please remember that each non-leaf node must have two children. It is not possible to have a node that has only one child node.

In this class, we will use a method called post-order travel with control of 1 or 0. This method can uniquely describe each code tree.

This is how it works. This is post-order traversal with slight changes. If a node is a leaf node, then 1 is printed before printing this node’s character. If a node is not a leaf node, print 0.

Please check this function. It is very similar to the original post-order traversal function described earlier. The only change is how each node is printed. If it is a leaf node, print one before printing the character.

Otherwise, print 0.

Let’s how this code tree will be described. We have seen this code tree before.

First, we write down the post-order traversal of the tree. Alpha, Beta, Gamma, Delta, and Epsilon are added to describe the non-leaf nodes.

The post-order traversal of this tree is

Ei, M, #, Gee, delta, gamma, beta, cee, ass, epsilon, alpha.

Next, we add one in front of each character in one of the leaf nodes. The Greek letters are replaced by 0.

This code tree can be described as

1, Ei, 1, M, 1 #, 1, Gee, 0, 0, 0, 1, cee, 1 ass, 0, 0.

Next, let’s convert the description back to the code tree.

The ones and zeros are the control. When we see one, we need to read the character after one, and add a tree node to the list.

When we see zero, we take the last two tree nodes, make them share a parent node, and add the parent node back.

Let’s go through the example from the description 1, Ei, 1, M, 1, #, 1, Gee, 0, 0, 0, 1, cee, 1, ass, 0, 0 to build the code tree.

We are going to use a square to represent a list node. An oval represents a tree node.

When we see one ei, we create a tree node that stores ei. This tree node is pointed by a list node.

Then, we see one M. . Another tree node is created to store M. . This tree node is pointed by the second list node.

We continue this process after seeing one # and one gee.

At this moment, we have four tree nodes storing the four characters. The four tree nodes are connected by a linked list of four list nodes.

Next, we see a zero. What we have seen so far is one, ei, one, M, one, #, one, gee, and zero.

We take the last two list nodes, and make the tree nodes siblings. The parent node is added back to the list. At this moment, the linked list has three list nodes. The last list node points to a tree node that has two children. The left child stores #. The right child stores gee.

The next zero takes the last two list nodes again. The linked list now has two list nodes. The last list node points to a tree node. This tree node has M as the left child. The right child of this tree node is the parent of # and gee.

We see another zero again. Now, there is only one list node pointing to a tree node that is the parent of ei and the tree node that was created earlier.

Next, the input is one cee and one ass. Two more list nodes are added. They point to two new tree nodes for storing characters cee and ass.

The next input is zero. This zero creates a new tree node that is the parent of cee and ass.

The next input is zero again. A new tree node is created as the parent.

If we compare this tree with the code tree earlier, we can see that this reconstructed tree is the same as the code tree. We have successfully rebuild the tree from the post-order description of

1, Ei, 1, M, 1, #, 1, Gee, 0, 0, 0, 1, cee, 1, ass, 0, 0.

To indicate the ending of the description, one more zero is added. After this ending error, nothing will be added to the tree.

Here are some frequently asked questions about reconstructing the code tree from the post-order description.

First, one and zero are used for control. Does that mean this method cannot handle one or zero in the character?

The answer is that this method can handle one or zero. Consider this example, one one creates a tree node with one as the character. One zero creates another tree node that stores zero. If the next is zero, these two tree nodes become siblings.

Another question is whether the code tree needs to store the occurrence? The answer is no. The occurrences are needed for creating the code tree during compression.

The occurrences are not needed for reconstructing the code tree during decompression.

The next question is whether this method will create a unique code tree. The answer is yes.

How many ones and zeros are needed? Each leaf node needs one. Each non-leaf node needs zero. If there are N leaf nodes, there will be N minus one non leaf nodes. Adding the final ending zero, there are N zeros.

What is the simplest tree? The simplest tree has nothing. If the input is empty. Nothing needs to be compressed.

If the input is one followed by a character and then zero, this is a tree of only one node. That means the input has only one type of character.

As mentioned earlier, the second part of the header is the length of the file. How is the length expressed? It is expressed as 4 bytes, or 32 bits. This can specify a number larger than four billion. It is larger than number of characters in any book that has ever been published.