This lecture explains how linked list and binary tree can be used together for data compression.

We will explain the Huffman compression.

Huffman compression is widely used. This lecture talks about how to build the compression tree and compress data.

Huffman compression is an example of variable length coding. What does that mean?

Consider Ass key coding, the American Standard Code for Information Interchange. Each character is encoded using 8 bits.

This is simple but somewhat wasteful.

In English, some characters such as ass and E are used often. Some other characters, such as X and Z, are used rarely. If we use variable length coding, the data can be expresses more efficiently. Efficiency is measured by the average number of bits per character.

If a character is used frequently, it uses fewer bits.

If a character is used rarely, it uses more bits.

On average, we need to use fewer bits for the data.

The general principle in system design is to optimize for frequent cases. This method is lossless compression. The original data can be fully recovered.

Why do we need to compress files? Where is compression used? Compression is used everywhere. When you take a photo, you are using compression. When you take a video clip, you are using compression. Image, video, and audio usually use lossy compression. Lossy compression means the original data cannot be fully recovered but this is usually acceptable because human can recognize image, video, or audio of lower quality.

Compression is used when download files.

Consider the situation that you pay a mobile phone plan that has limited data transmission, such as 3 giga bytes per month. You may want to compress data to reduce the amount of transmitted data. If a network is slow, you may want to compress data so that you can get the data faster. If a network is unstable, compressing data can reduce the amount of transmitted data so that the data can have better chance of being received.

This lecture explains the Huffman compression method. Huffman compression is lossless compression. That means the original data can be fully recovered. This method has four steps: In the first step, the occurrences of characters are counted. This step includes all symbols and the unprintable characters, if they appear in the original input.

The second step sorts the characters in the ascending order.

The third step takes the least two occurrences, makes them share the same parent node, makes the parent’s occurrence the sum of the children’s occurrences, and inserts this back in the ascending. This step removes two and adds one. Thus, this step effectively reduces the number of characters by one.

Continue the third step until only one node is left.

Let’s go through an example and understand how this works.

Consider an article that has these characters, #, ei, gee, cee, M, and ass.

Their occurrences are 4, 18, 7, 22, 10, 35 respectively.

These characters are sorted by their occurrences in the ascending order. # appears the least, only 4 times, and it is at the beginning. Ass appears the most, 35 times, and it is at the end.

Next, we take the first two and make them the siblings of a binary tree. Thus, # and gee are the left child and the right child of a binary tree. The parent’s occurrence is the sum of 4 and 7, equal to 11.

This parent is inserted back, still in the ascending order.

This process continues. We take the first two, make them the left and right children of a binary tree. The parent’s occurrence is the sum of 10 and 11. This is again inserted in the ascending order.

Each time, we take the first two, and insert one back. Thus, the length decreases by one.

This process continues. The first two include ei that occurs 18 times and this tree that occurs 21 times. They share the same parent and the total occurrence is 39. 39 is larger than 22 and 35. Thus, 39 is inserted to the end.

Cee and ass become the left and right children of a binary tree node. The occurrence of the parent is 22 plus 35 equal to 57. 57 is greater than 39. Thus, the newly created binary tree is inserted to the end.

Finally, this is only one node left. The total occurrence is 96. The left subtree includes ei, M, #, and gee. The right subtree includes cee and ass.

From this code tree, we know that only the leaf nodes contain characters. This is how the tree is constructed.

After building this tree, we can decide the code for the characters. From the root, going left means code of 0. Going right means code of 1.

From the root to character ei, we need to follow the left edge twice. Thus, the code of ei is 0, 0.

From the root to character M, we need to follow the left edge once, the right edge once, and the left edge again. Thus, the code of M is 0, 1, 0.

To reach # from the root, we need to go left, right, right, and left. The code for # is 0, 1, 1, 0.

From the root, we need to go left, right, right, right to gee. The code for gee is 0, 1, 1, 1.

To reach cee from the root, we need to go right and left. Thus, the code for cee is 1, 0.

Finally, to reach ass from the root, we need to go right and right. The code for ass is 1, 1.

From the table and based on the procedure of building the tree, we know the following properties are true. If a character X occurs fewer times than another character Y, then, the length of the code for X cannot be shorter than the length of the code of Y. .

Their lengths, however, may be the same.

The other direction is not true. If the length of the code for X is greater than the length of the code for Y, we cannot say that X occurs fewer times than Y. .

They may have the same occurrences.

This table is called the code book. It allows us to convert a character to the code using 0 or 1.

Next, we will see how to use the code book to convert an article composed of these characters.

Let us consider the input of ei, ei, cee, ass, #, M, gee, cee, ass, ei, etc. .

The output is the code of these characters, 0, 0 for ei; 0, 0 for ei; 1, 0 for cee; 1, 1 for ass; 0, 1, 1, 0 for #; 0, 1, 0 for M, etc. .

Next, we explain how to take the code and convert it back to the characters in the original input.

Consider the codes as the input and the code tree shown here.

First, we read 0. From the root, we move left.

Then, we read another zero and move left again. Now, we have reached a leaf node and it is ei. Thus, the first two zeros mean the character ei.

We have used the first two zeros for the first character ei.

We will go back to the root of the tree and read the code.

We see a zero and go to the left child of the root.

We see another zero and go to the left child of the root. We reach a leaf node. It is ei again. Thus, the first four zeros from the code give us two ei’s.

The next code is one. From the root, we move to the right child. The next code is zero. We move to the left child. This is a leaf node of cee. Thus, the next character is cee.

We go back to the root of the tree and read two ones. This is the code of ass.

So far, every character has the same length.

Next, we read 0 from the input and go to the left child. The next code is one and we go to the right child. This is not a leaf node yet. Thus, we keep reading.

The next is one again and we go to the right child. This is still not a leaf node.

We read the next code and it is zero. We reach a leaf node and it is #.

This slide summarizes how to take the codes of zeros and ones and convert the codes to the characters. If the code is zero, go to the left child. If the code is one, go to the right child. If we reach a leaf node, output the character of that leaf node. Go back to the root of the tree.

Please remember, characters are stored in the leaf nodes only. This is how the tree is created.

Next, let’s consider how to build the compression tree. You can get the sample code from this github repository.

We need two data structures, using both binary tree and linked list.

The tree node contains four attributes: pointers for left and right children, the character, and the occurrence.

The list node has two attributes: a pointer for the next list node, and a pointer for a tree node.

This is how the tree nodes and the list nodes will be used.

The characters and their occurrences are stored in the tree nodes. The list nodes point to the tree nodes in the ascending order of the occurrences.

The method takes the first two list nodes and get the first two tree nodes.

Create a new tree node as the parent. The occurrence is the sum of the two children’s occurrences.

A new list node is created to point to this tree node.

The list node is inserted into the list in the ascending order.

This slide shows the second iteration. The first two tree nodes are taken from the list. A new tree node is created as the parent of these two nodes. The new tree node’s occurrence is the sum of 10 and 11.

A new list node is created and points to this new tree node.

The list node is inserted to the list in the ascending order.