Henning Kjoeita

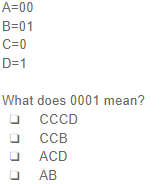
**Huffman Encoding**

Compression of data means we represent data in some way that takes less memory compared to the uncompressed version. Then from the compressed version we can decode it to get the original data. The challenge is to use the minimum amount of space needed while also not lose any data. Huffman encoding solves this by using something called prefix codes. Huffman does this very efficiently, in fact Huffman is an optimal algorithm (that encoding characters individually) so no wonder it is widely used.

Suppose there is only 7 characters and the 8th character is ‘space’, we can then encode everything in 3 bits. Normally characters are encoded using ASCII that uses one byte (8 bits) for each character, but for simplicity’s sake we pretend there is only 8 characters. We can then create codes for each character to represent them in bits and we may get the table below.

|  |  |
| --- | --- |
| Character | Binary |
| A | 000 |
| B | 001 |
| C | 010 |
| D | 011 |
| E | 100 |
| F | 101 |
| G | 110 |
| SPACE | 111 |

The text “BED FED CAB BAG DAB BAD BAE BEE” takes 93 (31\*3=93) bits to write in binary, there is 31 characters (including space) and each character takes 3 bits. Note that some characters are much more frequent, space and B both appear 7 times, while F and G only once. If we could write space and B with two bits, and F and G with four or five bits we would use less memory.

However, it is not that simple. The new codes would need to be able to be decoded back to the original text. Look on the question to the right, in the example 0001 can be

decoded to all the options. There is ambiguity because some codes are

prefixes of something else (0 is a prefix of 00 and 01), we want the new

codes to use less bits for more frequent characters and have no ambiguity.

In order to not have ambiguity we need to use prefix coeds, which is codes

where no complete code is a prefix of another code.

Huffman encoding does this by first building a tree and then traversing the

tree to get the codes for each character. To build the tree we use the steps

below.

1. Each letter is a leaf node, the weight of the letter is the frequency it appears.
2. Build a min-heap of the nodes (each node has a lower (or equal) weight than its children)
3. Take out the two next nodes
4. Add a new internal node, this nodes weight is equal to the sum of its two children
   1. First node we took out is the left child of the internal node
   2. Second node we took out is the right child of the internal node
5. Repeat step 3 and 4 until only one node is left in the heap.

Before we begin with figuring out the codes we count the number of times each character appears, but we can also use other values instead of the frequency as the weight. For example, the probability a character occurs can be used and we’d not need to read the whole text assuming the probability is known. Anyway, in “BED FED CAB BAG DAB BAD BAE BEE” we can see that:

SPACE appear 7 times

A appear 5 times

B appear 7 times

C appear 1 times

D appear 4 times

E appear 5 times

F appear 1 times

G appear 1 times

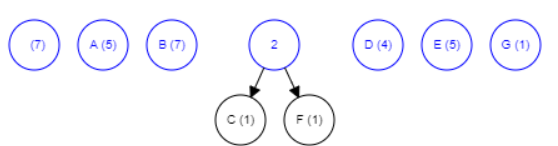
We put the characters into a min heap, here shown graphically using a tool created by The University of British Columbia.



From our heap we pull the first two nodes (C and F)

We create a new node (2) equal to C and F who’s left child is C and right child is F.

Then we add the new node back to the heap.

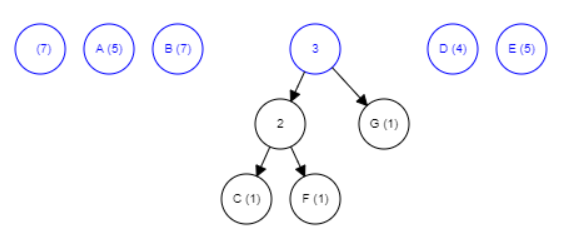


We repeat until there is one node left in the heap.

From our heap we pull the first two nodes (G and 2)

We create a new node (3) equal to G and 2 who’s left child is G and right child is 2.

Then we add the new node back to the heap.

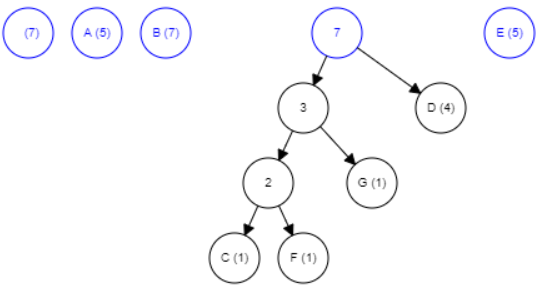


We repeat until there is one node left in the heap.

From our heap we pull the first two nodes (3 and D)

We create a new node (7) equal to 3 and D who’s left child is 3 and right child is D.

Then we add the new node back to the heap.

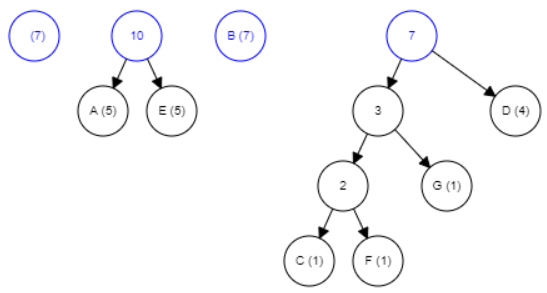


We repeat until there is one node left in the heap.

From our heap we pull the first two nodes (A and E)

We create a new node (10) equal to E and A who’s left child is A and right child is E.

Then we add the new node back to the heap.

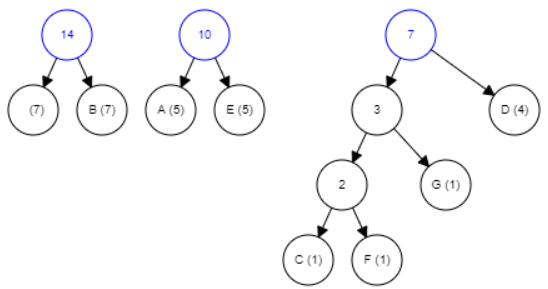


We repeat until there is one node left in the heap.

From our heap we pull the first two nodes (SPACE and B)

We create a new node (14) equal to SPACE and B who’s left child is SPACE and right child is B.

Then we add the new node back to the heap.

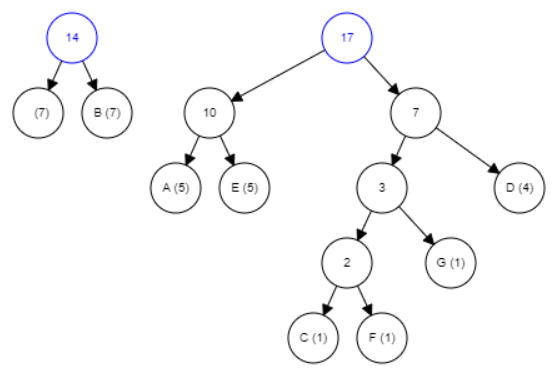


We repeat until there is one node left in the heap.

From our heap we pull the first two nodes (7 and 10)

We create a new node (17) equal to 7 and 10 who’s left child is SPACE and right child is 10.

Then we add the new node back to the heap.



We repeat until there is one node left in the heap.

From our heap we pull the first two nodes (14 and 17)

We create a new node (31) equal to 13 and 17 who’s left child is 14 and right child is 17.

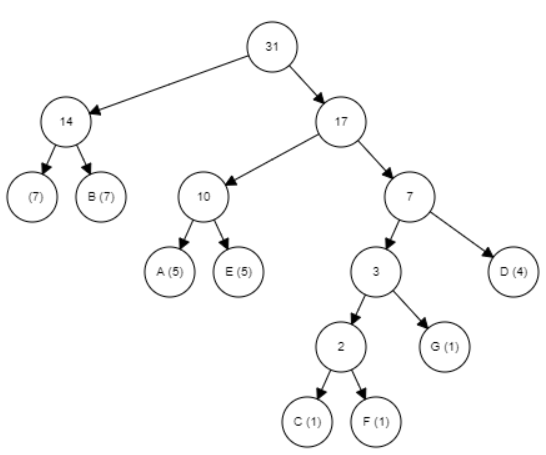
Then we add the new node back to the heap.

At this point there is only one node left in the heap so we stop.

The last remaining step is to traverse the tree, traversing the tree follows two simple rules.

If we go left then add a 0, if we go right then add a 1.

Every time we reach a leaf then we have the code for that leaf.



Here is the complete tree, to find the code for B we begin at the root. First, we go left in the tree (add 0) then we go right (from 14) in the tree (add 1). The code for B is 01.

By traversing to every leaf and we can create a table of codes, for this tree the binary codes are:

|  |  |  |  |
| --- | --- | --- | --- |
| Character | Binary | Huffman Encoding | Frequency |
| A | 000 | 101 | 5 |
| B | 001 | 01 | 7 |
| C | 010 | 11000 | 1 |
| D | 011 | 111 | 4 |
| E | 100 | 101 | 5 |
| F | 101 | 11001 | 1 |
| G | 110 | 1101 | 1 |
| SPACE | 111 | 00 | 7 |

Notice that the more frequent characters have shorter codes with Huffman Encoding. In this example with Huffman Encoding we used 9 bits less (Huffman encoding takes 84 bits, binary takes 93), or about 10% less. Also note that there can be multiple optimal Huffman codes, for example my implementation creates slightly different codes for the same text and still use the same amount of space. That is because some values are equal so variation can happen (the order in which characters are added to the heap matters when their values are equal). My implementation is attached at the end.

Since the numbers of characters are usually a small number the time complexity is not a big concern with Huffman encoding. It does not necessary matter what the complexity is if n is always a small number. Traversing the Huffman tree we have to visit each node once, so traversing the tree to find the codes is of a complexity O(n). However, we also have to first build the tree so and that’s the ‘hard’ part. We can put all the nodes in an array and build a heap from the array- this takes O(n) time using Buildheap algorithm. Then to create the tree we take out two nodes, and add one node. Insertion/ taking out a node takes log n time. We have to repeat that n-1 times (1 less in the heap each time, repeat until only 1 node in the heap.) (n-1)\*(log n+log n+log n)= O(n log n). In other words the ideal time complexity for huffman encoding is O(n log n). Note that reading a file to get the frequencies are not part of this, and would increase time quite a bit- especially if the file is long. That can be avoided by using excepted frequencies of letters rather than the actual frequencies.

The algorithm will always produce a valid prefix tree, as each time two subtrees are merged the left gets a 0 as their prefix and the right 1- and therefore their codes will always be valid prefix codes. In an optimal compression the symbols that have higher weight (frequency) will have shorter codes than symbols with lower weight. This is intuitive, as otherwise the codes between a less frequent character and a more frequent character could be switched to have a better compression. Huffman Encoding always gives the more frequent characters shorter code words (through the use of a min heap), and as such it must be optimal.

Additionally, in an optimal prefix compression the two characters with the lowest weight will have the same length. Suppose the code for A is 0100 and the code for B is 0101XXXX, where X represent more bits. Knowing that this is prefix codes, we can drop the X bits and b will still be distinct from A (as otherwise A would be a prefix of B). 0101 will also not be a prefix of anything else, because B and A are the two lowest weight characters. As such, an optimal compression the two lowest weight characters will always be coded with the same length. We can observe that this is also the case in Huffman Encoding, and therefore it must be optimal.

Compression means taking up less memory, which is useful when storing files. The more files stored (or larger files) the more important compression is, simply put using less memory is great. Additionally, when transferring data, especially over a network then less data to transfer speeds up the process of transferring data. Huffman Encoding is used in many compression tools, for example ZIP, JPEG, PNG and MP3 all use Huffman Encoding (in addition to some pre-work before running Huffman). Huffman Encoding us used all the time, by everyone, even though they don’t realize it.

My implementation in java, can also be found on GitHub here:

<https://github.com/Hennns/Huffman-Encoding/tree/master/src>

**import** java.io.File;

**import** java.io.FileNotFoundException;

**import** java.util.ArrayList;

**import** java.util.Arrays;

**import** java.util.Comparator;

**import** java.util.PriorityQueue;

**import** java.util.Scanner;

**public** **class** Main {

**static** **int** *totalBitCounterHuffman* = 0;

**static** **int** *totalBitCounterText* = 0;

**public** **static** **void** main(String[] args) {

**final** **int** bitsForTextChar = 3; // set this value to 8 for the first test

//first test file

// ArrayList<Tracker> data = readFile("src/The Taste of Melon.txt");

// Second test file

ArrayList<Tracker> data = *readFile*("src/test.txt");

**char**[] charArray = **new** **char**[data.size()];

**int**[] charfreq = **new** **int**[data.size()];

**for** (**int** i = 0; i < data.size(); i++) {

charArray[i] = data.get(i).getCharacter();

charfreq[i] = data.get(i).getCounter();

}

// These are used for testing

// char[] charArray = { 'a', 'b', 'c', 'd', 'e', 'f'};

// int[] charfreq = { 12, 2, 7, 13, 14, 85};

**int** length = charArray.length;

PriorityQueue<Node> heap = **new** PriorityQueue<Node>(length, **new** CompareNode());

// this creates a heap

**for** (**int** i = 0; i < length; i++) {

*totalBitCounterText* += charfreq[i] \* bitsForTextChar;

System.***out***.println(charArray[i] + " appear " + charfreq[i] + " times");

Node nextNode = **new** Node(charArray[i], charfreq[i]);

heap.add(nextNode);

}

**while** (heap.size() > 1) {

Node nodeA = heap.poll();

Node nodeB = heap.poll();

Node root = **new** Node((nodeA.getWeight() + nodeB.getWeight()), nodeA, nodeB);

System.***out***.println(root.getWeight());

heap.add(root);

}

System.***out***.println("Letter | Code");

// The heap is empty after this line, the last Node is the root node.

*printCodes*(heap.poll(), "");

System.***out***.println("Assuming each char takes " + bitsForTextChar + " bits");

System.***out***.println("The text was " + *totalBitCounterText* + " bits");

System.***out***.println("With Huffman Encoding we use " + *totalBitCounterHuffman* + " bits");

**float** percent = (*totalBitCounterText* - *totalBitCounterHuffman*) \* 100f / (*totalBitCounterText*);

System.***out***.println("That is " + (*totalBitCounterText* - *totalBitCounterHuffman*) + " bits less");

System.***out***.println("That is " + percent + " % decrease");

}

// traverse the tree to find the codes

**private** **static** **void** printCodes(Node n, String code) {

**if** (n.isLeaf()) {

System.***out***.println("'" + n.getCharacter() + "'" + code);

// System.out.println((int)n.getCharacter());

*totalBitCounterHuffman* += code.length() \* n.getWeight();

// System.out.println("'" + n.getCharacter() + "' takes up

// "+n.getWeight()\*code.length()+" bits");

**return**;

}

*printCodes*(n.getLeftChild(), code + "0");

*printCodes*(n.getRigthChild(), code + "1");

}

// reads given file

**private** **static** ArrayList<Tracker> readFile(String path) {

File file = **new** File(path);

ArrayList<Tracker> in = **new** ArrayList<Tracker>();

String s = "";

**try** {

Scanner sc = **new** Scanner(file);

s = sc.useDelimiter("\\Z").next();

sc.close();

} **catch** (FileNotFoundException e) {

e.printStackTrace();

}

// We put it as an array of characters to easier sort and count them

**char** tempArray[] = s.toCharArray();

Arrays.*sort*(tempArray);

// add the first character so we can compare to something

in.add(**new** Tracker(tempArray[0]));

**int** counter = 0;

// we start looping at 1 since we already added the first character

**for** (**int** t = 1; t < tempArray.length; t++) {

**if** (in.get(counter).getCharacter() == (tempArray[t])) {

in.get(counter).increment();

// System.out.println(in.get(counter).getCharacter() + " have

// appeared " + in.get(counter).getCounter() + " times");

} **else** {

// System.out.println("added "+tempArray[t]);

in.add(**new** Tracker(tempArray[t]));

counter++;

}

}

**return** in;

}

}

// we use this to compare the Nodes in the heap

**class** CompareNode **implements** Comparator<Node> {

**public** **int** compare(Node x, Node y) {

**return** x.getWeight() - y.getWeight();

}

}

**class** Tracker {

**private** **char** c;

**private** **int** counter;

**public** Tracker(**char** letter) {

c = letter;

counter = 1;

}

**public** **void** increment() {

counter++;

}

**public** **int** getCounter() {

**return** counter;

}

**public** **char** getCharacter() {

**return** c;

}

}

**public** **class** Node {

**private** **final** **int** weigth;

**private** **final** **char** c;

**private** **final** Node leftChild;

**private** **final** Node rigthChild;

**public** Node(**char** letter, **int** value) {

c = letter;

weigth = value;

leftChild = **null**;

rigthChild = **null**;

}

**public** Node(**int** value, Node left, Node rigth) {

c = '\0';

weigth = value;

leftChild = left;

rigthChild = rigth;

}

**public** **int** getWeight() {

**return** weigth;

}

**public** **char** getCharacter() {

**return** c;

}

**public** Node getRigthChild() {

**return** rigthChild;

}

**public** Node getLeftChild() {

**return** leftChild;

}

**public** **boolean** isLeaf() {

**if** (leftChild == **null** && rigthChild == **null**) {

**return** **true**;

}

**return** **false**;

}

}

Test.txt:

BED FED CAB BAG DAB BAD BAE BEE