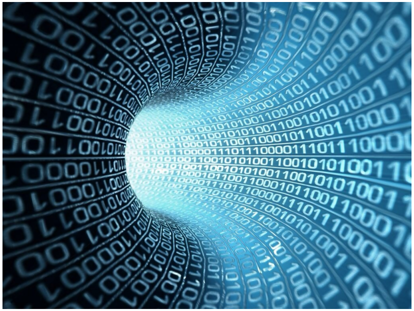
CI583: Data Structures and Operating Systems
Compression algorithms



Outline

Data, data everywhere!

In the digital age, we're drowning in data. Google pioneered the "Big Data" era by never throwing anything anyway and storing loosely connected or unconnected data with the aim of applying meaning to it later on.

Almost all data generated is metadata produced automatically.

The global data volume is estimated to have exceeded 64 zettabytes in 2020.

Data, data everywhere!

It's hard to imagine how much data is contained in 64 zettabytes...

1 zettabyte = 1000 exabyte = 1 million terabytes = 1 trillion GB

It has been estimated that all human words ever spoken could be encoded into 5 exabytes[1].

[1] New York Times: `http://tx0.org/5ls`

Compression algorithms

Storage might be relatively cheap but the fact that we can store so much data and transfer it across networks etc is largely thanks to effective and widely used compression algorithms which retain, to a greater or lesser degree, the meaning of the input.

The field that studies this is called Information Theory, combining mathematics and computer science, and founded by Claude Shannon with a series of landmark works in the 1940s and 50s.

A compression algorithm can be lossless (information-preserving) or lossy (some information considered non-essential is discarded).

Run-length encoding

Perhaps the simplest example of a lossless encoding is a run-length encoding (RLE). This works by replacing repetition with a single value and a count. Thus, if we have a protocol that encodes the pixel values of one row of a black-and-white image like so:

```
WWWWWWWWWWWWBWWWWWWWWWWWWBBBWWWWWWWWWWWWWWWWWWWWWW
WWBWWWWWWWWWWWWWW
```

this has the following RLE:

```
12W1B12W3B24W1B14W
```

Optimising RLE

RLE algorithms are simple and can be extremely effective especially for data with little variation, where savings of up to 90% are not unusual.

To store data with more variation in it, such as text, we can optimise the process by transforming the input before applying the RLE. Obviously, such a transformation needs to be reversible so that we can recover the original.

Burrows-Wheeler Transform

The Burrows-Wheeler Transform (BWT) transforms a sequence of characters so that the most commonly repeating characters are adjacent to each other, making the application of RLE more effective. That would be easy enough in itself – just sort the input – but, marvellously enough, BWT is also reversible.

This transformation is used in the `bzip` compression format, widely used on UNIX systems.

Burrows-Wheeler Transform

Take some input, e.g. the word `billing`. To optimise this for RLE we want multiple occurrences of the same character to be next to each other. Put special characters at the start and end of the input, e.g. `^billing$`.

Form the set of rotations of the input. We rotate a word by moving one character from the beginning to end.

Burrows-Wheeler Transform

```
^billing$
billing$^
illing$^b
lling$^bi
ling$^bil
ing$^bill
ng$^billi
g$^billin
$^billing
```

Then sort this table in lexicographic order, where `x` < `^` < `$`.

Burrows-Wheeler Transform

`billing$^`
`g$^billin`
`illing$^b`
`ing$^bill`
`ling$^bil`
`lling$^bi`
`ng$^billi`
`^billing$`
`$^billing`

Then the last column of the table is the BWT of the input: `^nbllii$g`.

Burrows-Wheeler Transform

To take the inverse BWT, i.e. to recover the original, we can insert the BWT as one column of a table, then sort the rows of the table, add another column, sort, and so on. When we have added all columns and sorted for the last time, the row which ends with $ is the original input. Try it yourself with a short string.

As well as being used with “regular” text files, BWT is often applied to large amounts of data in bioinformatics – e.g. sequences of millions of characters or bits representing DNA.

Huffman coding

In textual data, each character typically takes 8 (ASCII) or 16 (UTF-16) bits. Clearly, we should be able to save some space just by encoding text in a binary format.

Huffman coding is an elegant compression algorithm that combines ideas from RLE (i.e. it exploits frequency information) with binary encoding to save space. Huffman coding was invented by David Huffman in 1952.

Huffman coding

The idea is to generate a binary sequence that represents each character required. This might be the English alphabet, some subset of that, or any collection of symbols. Say we have a 100KB file made up of repetitions of the letters `a` to `f`.

We start by creating a frequency table:

	a	b	c	d	e	f
Frequency (thousands)	45	13	12	16	9	5

Huffman coding

If we use a fixed-length code we can encode this data in about 37.5KB. If we use a variable-length code and assign the shortest code to the most frequently used characters, we can encode it in just 28KB.

	a	b	c	d	e	f
Frequency (thousands)	45	13	12	16	9	5
Code (fixed-length)	000	001	010	011	100	101
Code (variable-length)	0	101	100	111	1101	1100

Huffman coding

To send this data over the wire or store it in a file, we need to send/store the mapping from code to characters followed by the concatenation of all binary codes as they appear in the unencoded document.

Any sequence of codes must be unambiguous – we can only decode this at the other end if no code is a prefix of any other. If we had used `1` for `c` and `11` for `d`, how would we decode `111`?

Huffman coding

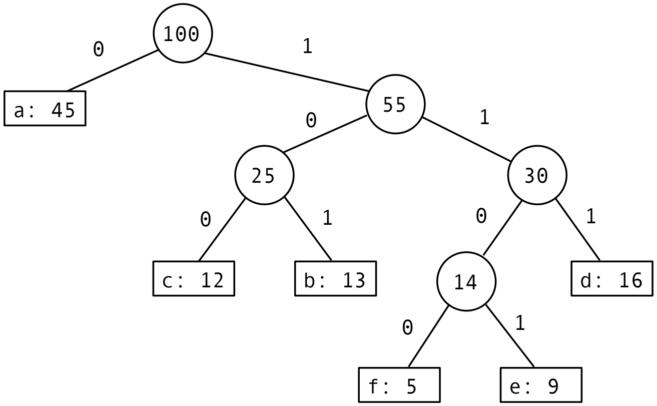
To send this data over the wire or store it in a file, we need to send/store the mapping from code to characters followed by the concatenation of all binary codes as they appear in the unencoded document.

Any sequence of codes must be unambiguous – we can only decode this at the other end if no code is a prefix of any other. If we had used `1` for `c` and `11` for `d`, how would we decode `111`?

	a	b	c	d	e	f
Code (variable-length)	0	101	100	111	1101	1100

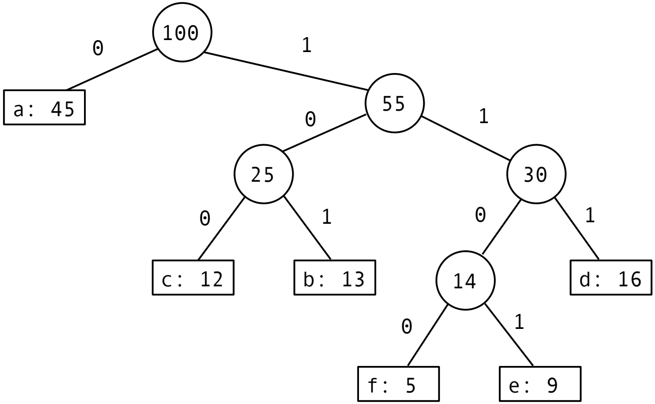
Huffman trees

We can create these variable-length codes using a binary tree (not a search tree). In a Huffman Tree the leaves contain the data, a character and its frequency. Internal nodes are labelled with the combined frequencies of their children.



Huffman trees

To decode data we go start at the root and go left for 0, and right for 1 until we get to a leaf. So, to decode 0101100, we start at the root, read an a, start at the root again, and so on to read abc. An optimal Huffman tree is full.

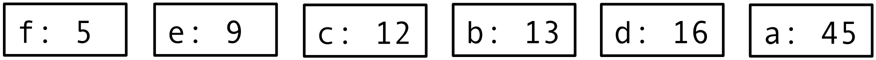


Creating the Huffman coding

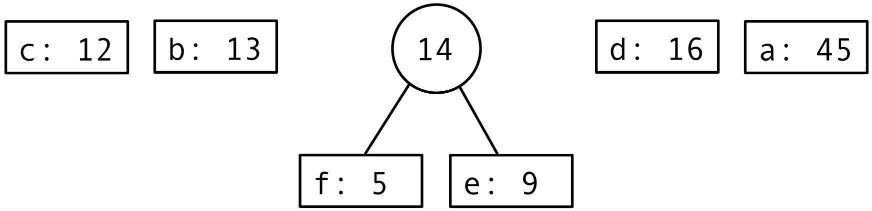
The most elegant part of this scheme is the algorithm used to create the tree:

1. Make a tree node object for each character, with an extra label for its frequency.
2. Put these nodes in a priority queue, where the lowest frequency has highest priority.
3. Repeatedly:
    1. Remove two nodes from the queue and insert them as children to a new node. The char label of the new node is blank and the frequency label is the sum of the labels of its children.
    2. Put the new node back in the queue.
    3. When there is only one item in the queue, that's the Huffman tree.

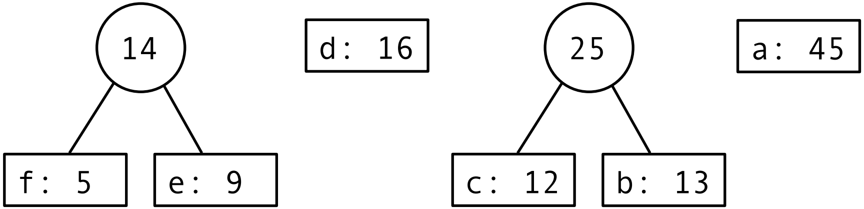
Creating the Huffman coding



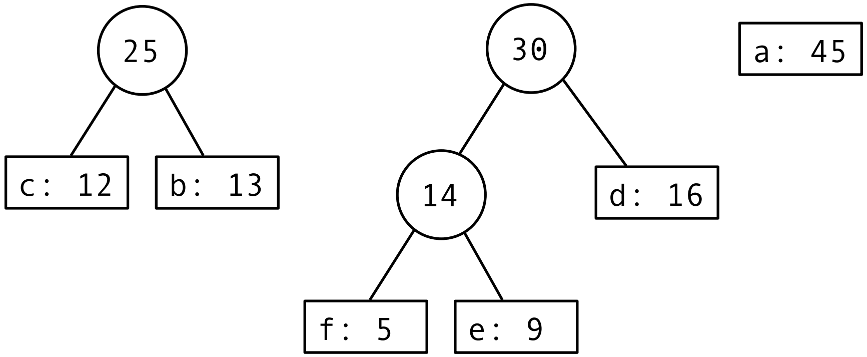
Creating the Huffman coding



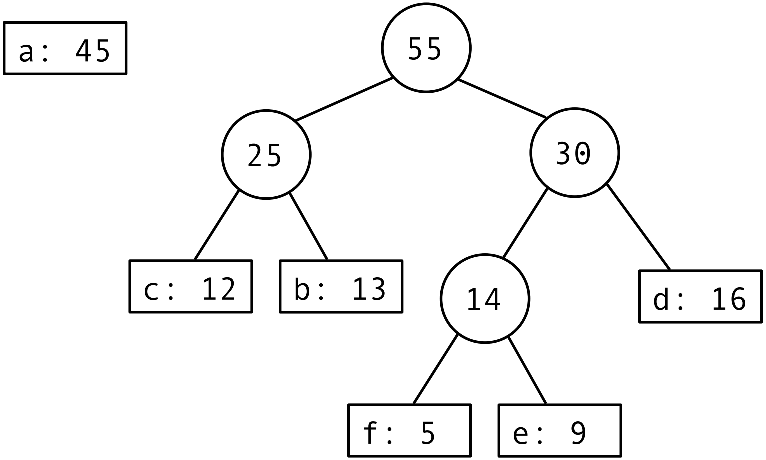
Creating the Huffman coding



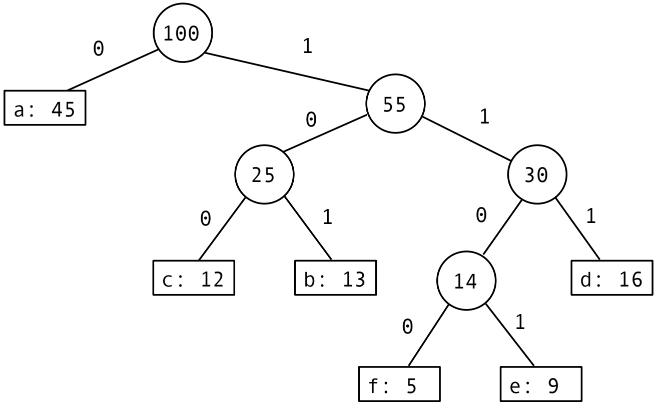
Creating the Huffman coding



Creating the Huffman coding



Creating the Huffman coding



Transmitting the Huffman coding

To use a Huffman coding as part of a compressed file format we begin the file with a “magic number” identifying the format used and the length of the alphabet, A, used by the file. We then store the table as an array in the next $|A|$ bits, followed by one code after another until the EOF character.

This scheme is used by some of the most widely used compressed formats such as GIF and ZIP.