
University of St.Gallen
DEFLATE
St.Gallen, 21. October 2024
Jonas Länzlinger
From insight to impact.

Agenda

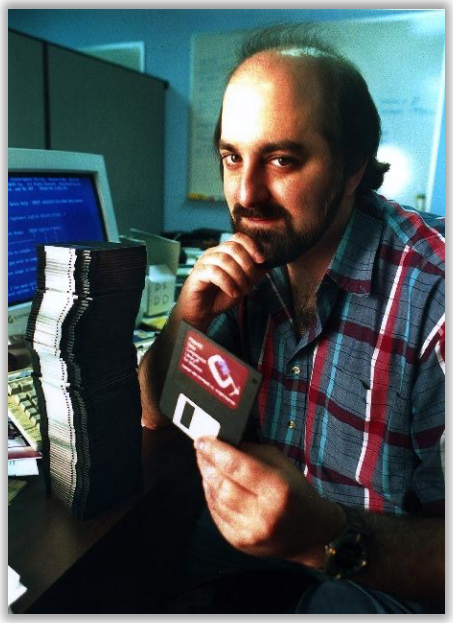
- What is *deflate*?
  - Background
  - Bird-eye view
- LZ77
  - What is it?
  - How does it work?
  - Examples
  - Challenges
- Huffman coding
  - How does it work?
  - Examples
  - Challenges
  - Canonical Huffman Codes
    - Construction / Deconstruction
- *Deflate*
  - In-depth view
  - Basic element – Blocks!
  - Non-compressed block
  - Static Huffman comp. block
    - Litlen codebook
    - Dist codebook
  - Dynamic Huffman comp. block
    - Codelen decoder
  - Why use different block types?
- Questions

What is *deflate*?

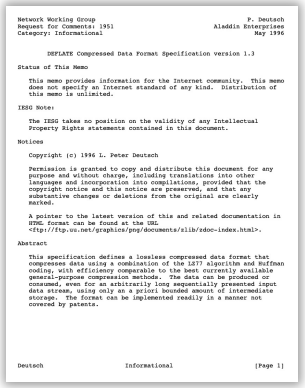
Background

- File format
- Combination of:
  - LZ77 algorithm
  - Huffman encoding
- Phil Katz 1993
- PKZIP archiving tool
- Patented, specified in RFC1951 (1996)

- Used in:
  - File compression
  - Compression of web content
  - PNG image compression
  - PDF compression
  - Open Document Formats (ODF)



RFC1951 (1996)



Network Working Group P. Deutsch
Request for Comments: 1951 Aladdin Enterprises
Category: Informational May 1996

DEFLATE Compressed Data Format Specification version 1.3

Status of This Memo

This memo provides information for the Internet community. This memo does not specify an Internet standard of any kind. Distribution of this memo is unlimited.

IESG Note:

The IESG takes no position on the validity of any Intellectual Property Rights statements contained in this document.

Notices

Copyright (c) 1996 L. Peter Deutsch

Permission is granted to copy and distribute this document for any purpose and without charge, including translations into other languages and incorporation into compilations, provided that the copyright notice and this notice are preserved, and that any substantive changes or deletions from the original are clearly marked.

A pointer to the latest version of this and related documentation in HTML format can be found at the URL <ftp://ftp.uu.net/graphics/png/documents/zlib/zdoc-index.html>.

Abstract

This specification defines a lossless compressed data format that compresses data using a combination of the LZ77 algorithm and Huffman coding, with efficiency comparable to the best currently available general-purpose compression methods. The data can be produced or consumed, even for an arbitrarily long sequentially presented input data stream, using only an a priori bounded amount of intermediate storage. The format can be implemented readily in a manner not covered by patents.

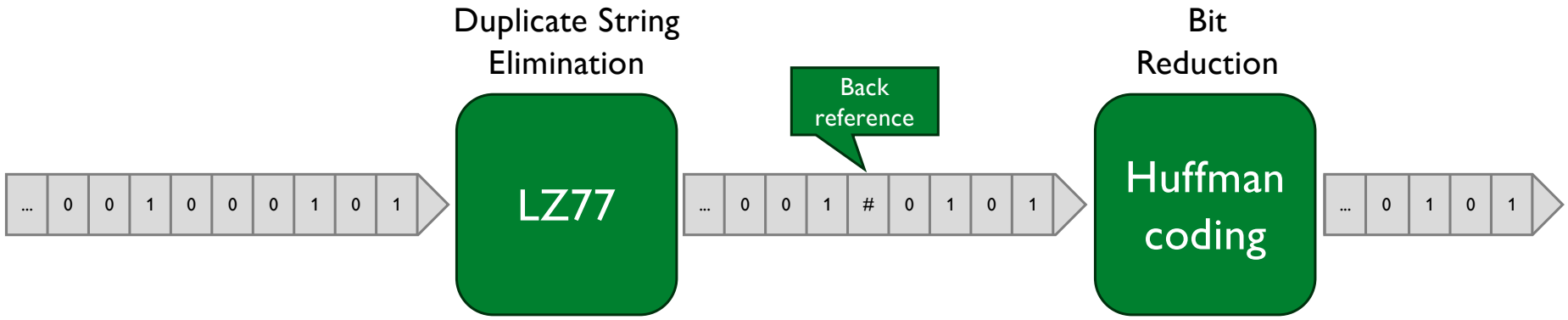
Deutsch Informational [Page 1]

"...*deflate* is a **lossless data compression** file format that compresses data using a combination of the **LZ77 algorithm and Huffman coding**, which **efficiency comparable to the best** currently available general-purpose compression methods. The data can be **produced or consumed**, even for an **arbitrarily long sequentially presented input data stream**, using only an **a priori bounded amount of intermediate storage**."

RFC1951: https://www.rfc-editor.org

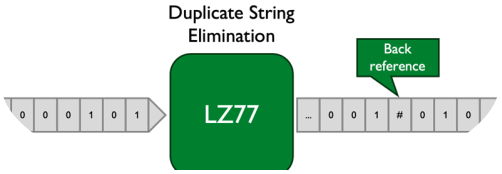
What is *deflate*?

Bird-eye view



LZ77

What is it?



- Lossless compression algorithm
- Lempel and Ziv in 1977
- Replacement of duplicates
- Triplets
- Dictionary-based compression

Triplet / Pointer / Back reference

[distance, length, next]

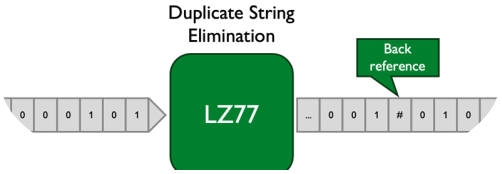
[distance, length]

Used in *deflate*

LZ77: https://learn.microsoft.com

LZ77

How does it work?



Terminology

- Input stream
- Byte
- Coding position
- Lookahead buffer
- Window (sliding)
- Pointer
- Match

1. Coding position to beginning of input stream.

2. Find longest match between window and lookahead buffer.

3. If n-match found: Output pointer + move coding position and window n-bytes forward.

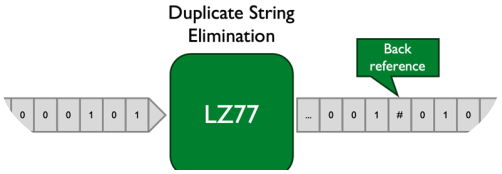
4. If no match found: Output null pointer and first byte in lookahead buffer + move coding position and window 1 byte forward.

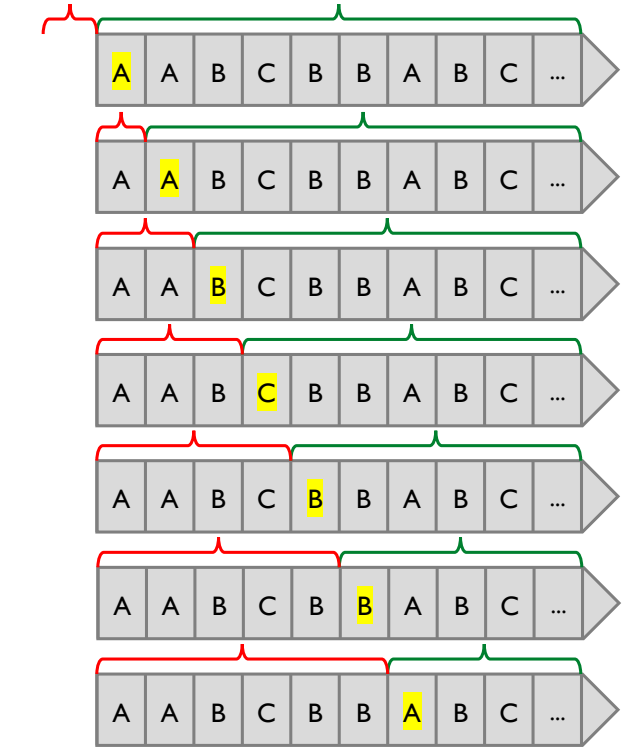
5. If lookahead buffer is non-empty, return to step 2.

LZ77: https://learn.microsoft.com
LZ77 Tutorial: https://medium.com

LZ77

Example 1





Output

(0,0)A

(0,0)A(1,1)

(0,0)A(1,1)(0,0)B

(0,0)A(1,1)(0,0)B(0,0)C

(0,0)A(1,1)(0,0)B(0,0)C(2,1)

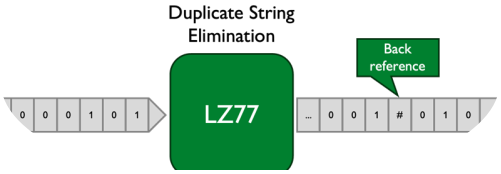
(0,0)A(1,1)(0,0)B(0,0)C(2,1)(1,1)

(0,0)A(1,1)(0,0)B(0,0)C(2,1)(1,1)(5,3)

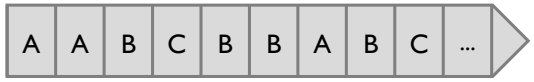
LZ77: https://learn.microsoft.com

LZ77

Example 1



Output	Decompressed
(0,0)A	A
(0,0)A(1,1)	AA
(0,0)A(1,1)(0,0)B	AAB
(0,0)A(1,1)(0,0)B(0,0)C	AABC
(0,0)A(1,1)(0,0)B(0,0)C(2,1)	AABCB
(0,0)A(1,1)(0,0)B(0,0)C(2,1)(1,1)	AABCBB
(0,0)A(1,1)(0,0)B(0,0)C(2,1)(1,1)(5,3)	AABCBBABC

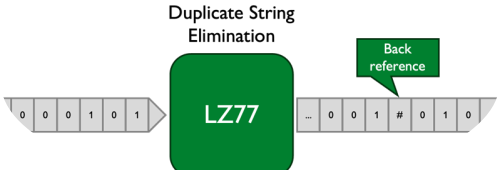


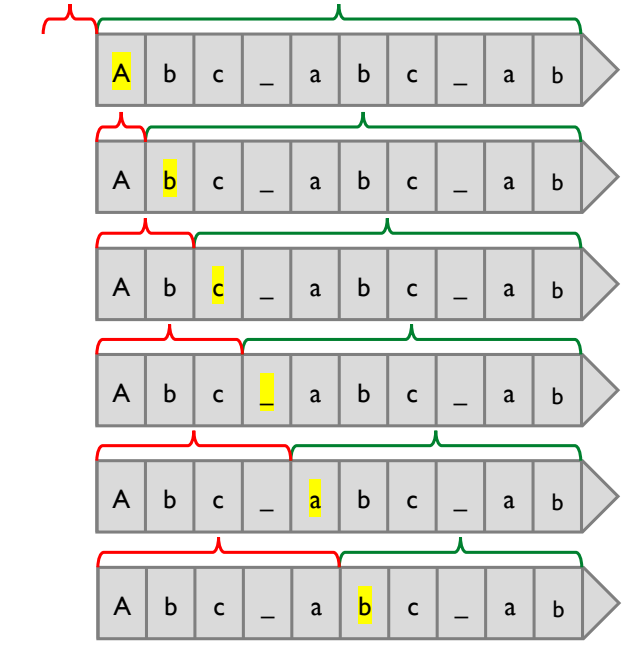
For efficiency reasons, only consider match, if:
len(match) > 3 bytes

LZ77: https://learn.microsoft.com

LZ77

Example 2





Output

(0,0)A

(0,0)A(0,0)b

(0,0)A(0,0)b(0,0)c

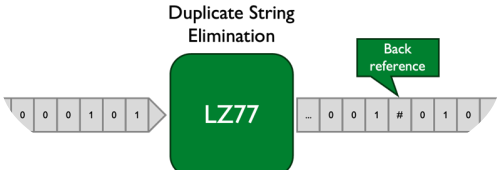
(0,0)A(0,0)b(0,0)c(0,0)_

(0,0)A(0,0)b(0,0)c(0,0)_(0,0)a

(0,0)A(0,0)b(0,0)c(0,0)_(0,0)a(4,5)

Overlap Example: https://github.com

LZ77
Challenges



"**How** do we find the **optimal matches**?"

Deflate suggested in RFC1951 a "hash-table based" approach:

- Maintain a hash-table
- Store offsets of 3-byte prefixes of the sliding window
- Whole sliding window is stored in-memory
- Match by prefixes, find longest match

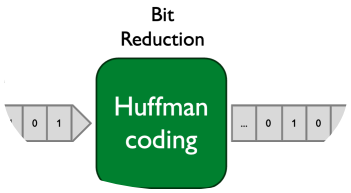
"**Open** field of **research**."

"Computationally **most expensive** step in ***deflate***."

LZ77 (tree based) implementation: https://github.com

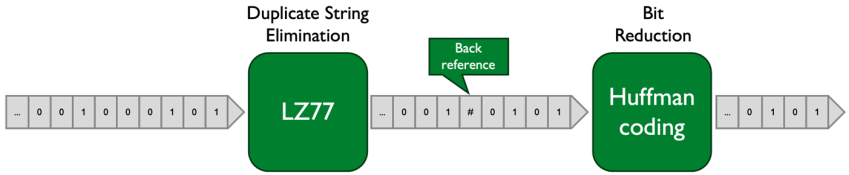
Huffman Coding

How does it work?



Bit Reduction Step

- Further reduce the LZ77 output by applying Huffman coding

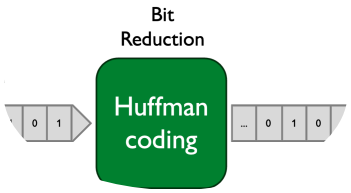


Huffman Coding

- Lossless compression
- Prefix-free code
- Variable-length code
  - Frequent symbols: short codes
  - Rare symbols: long codes
- Optimal code to this day
- Utilizing Huffman trees

Huffman Coding

Example Overview



Step 1. Symbol-wise count number of occurances.

Step 2: Bring table in ascending order of occurances.

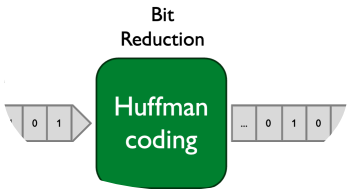
Step 3: From left, merge 2 symbols to one node with combined occurances.

Step 4: Repeat *step 3* until all symbols have been merged.

Step 5: Assign left axes the binary “0”; right axes the binary “1”.

Huffman Coding

Example Step 1+2



Step 1. Symbol-wise count number of occurances.

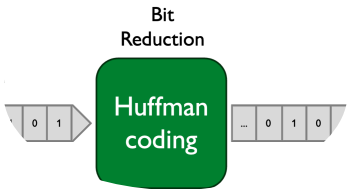
	A	B	C	D	Total
Occurances	2	3	4	1	10

Step 2: Bring table in ascending order of occurances.

	D	A	B	C	Total
Occurances	1	2	3	4	10

Huffman Coding

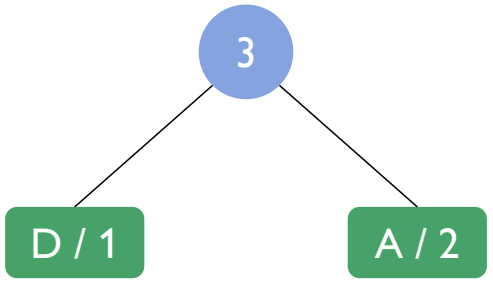
Example Step 3



Step 3: From left, merge 2 symbols to one node with combined occurances.

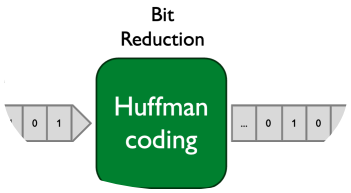
	D	A	B	C	Total
Occurances	1	2	3	4	10

	Node	B	C	Total
Occurances	3	3	4	10



Huffman Coding

Example Step 3+4

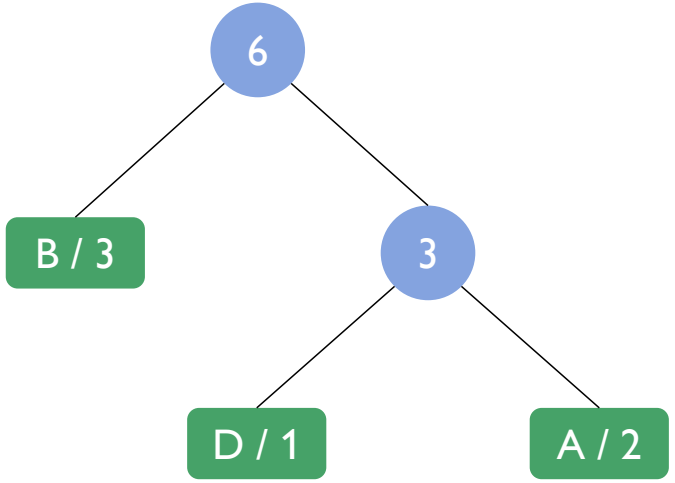


Step 3: From left, merge 2 symbols to one node with combined occurances.

Step 4: Repeat *step 3* until all symbols have been merged.

	Node	B	C	Total
Occurances	3	3	4	10

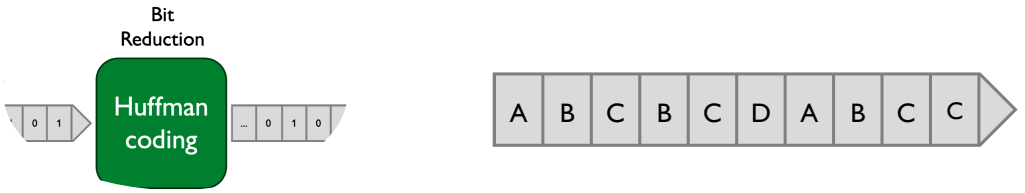
	Node	C	Total
Occurances	6	4	10



Resources: https://github.com

Huffman Coding

Example Step 3+4

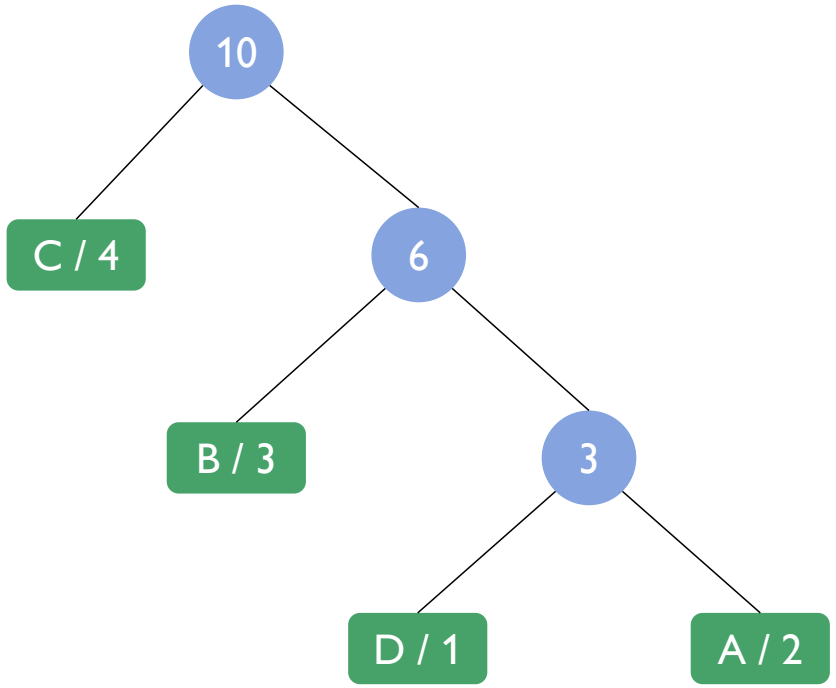


Step 3: From left, merge 2 symbols to one node with combined occurances.

Step 4: Repeat *step 3* until all symbols have been merged.

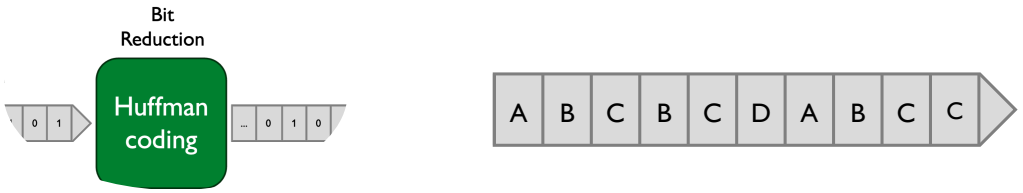
	Node	C	Total
Occurances	6	4	10

	Node	Total
Occurances	10	10



Huffman Coding

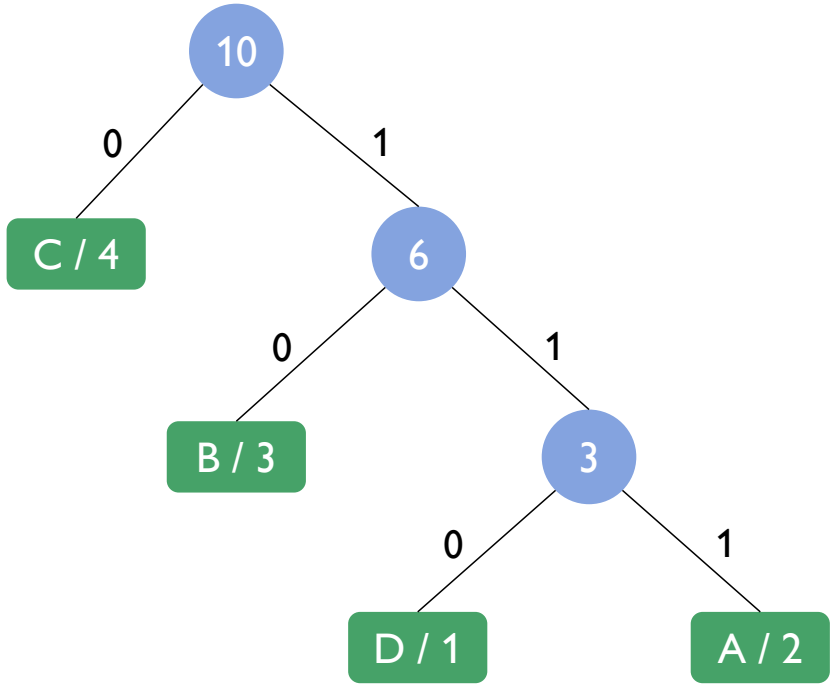
Example Step 5



Step 5: Assign left axes the binary “0”; right axes the binary “1”.

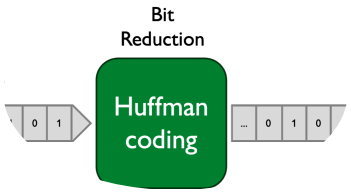
Codes

A	111
B	10
C	0
D	110



Huffman Coding

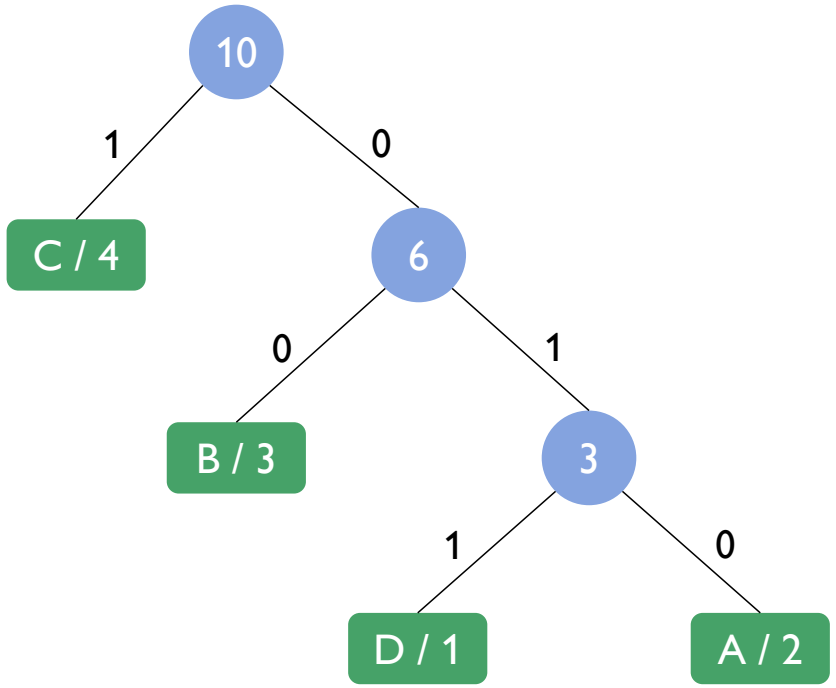
Challenge



What if we change the axes?

Codes	
A	111
B	10
C	0
D	110

Alt. Codes	
A	010
B	00
C	1
D	011



Resources: https://github.com

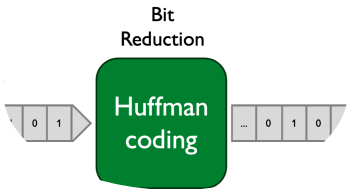
Huffman Coding

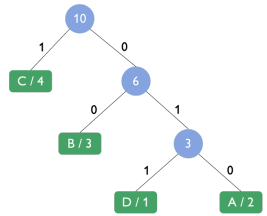
Canonical Huffman Codes

Codes		Alt. Codes	
A	111	A	010
B	10	B	00
C	0	C	1
D	110	D	011

Canonical Huffman Codes

Assigns codes to symbols in a pre-defined way, such that the individual **code lengths** lead deterministically to **only one** Huffman tree.





Rules

The DEFLATE specification cryptically defines it:

> The Huffman codes used for each alphabet in "deflate" format have two
>
> * All codes of a given bit length have lexicographically consecuti
> symbols they represet;
>
> * Shorter codes lexicographically precede longer codes.

It would be nice if they would avoid words like lexicographically but you can't have everything. You can also get confused over the term codes verses the binary values of the bytes in the alphabet. And of course shorter refers to bit count. That

- Th … ng the alp …
- Cc … mbol.
- Th … ng ordered.
- Th … s.

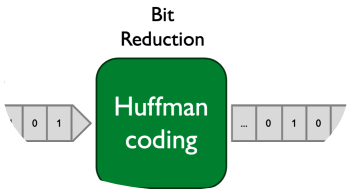
Codes

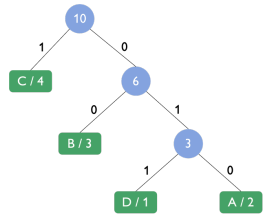
A	111	3
B	10	2
C	0	1
D	110	3

Resources: https://github.com

Huffman Coding

Canonical Huffman Codes





Codes	
A	111
B	10
C	0
D	110

Alt. Codes	
A	010
B	00
C	1
D	011

Canonical Huffman Codes

Assigns codes to symbols in a pre-defined way, such that the individual **code lengths** lead deterministically to **only one** Huffman tree.

Rules

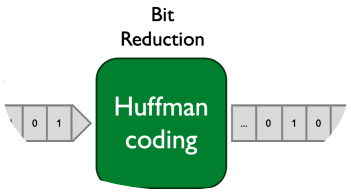
- The order of the code lengths are representing the alphabetically ordered symbols.
- Code length "0" represents a non-occuring symbol.
- The code words of a given length are ascending ordered.
- The shortest code word consists of only zeros.

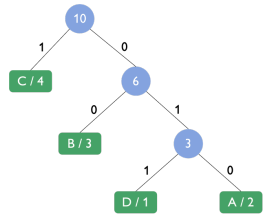
Alt. Codes

A	010	3
B	00	2
C	1	1
D	011	3

Huffman Coding

Canonical Huffman Codes





Codes		Alt. Codes	
A	111	A	010
B	10	B	00
C	0	C	1
D	110	D	011

Canonical Huffman Codes

Assigns codes to symbols in a pre-defined way, such that the individual **code lengths** lead deterministically to **only one** Huffman tree.

Rules

- The order of the code lengths are representing the alphabetically ordered symbols.
- Code length "0" represents a non-occuring symbol.
- The code words of a given length are ascending ordered.
- The shortest code word consists of only zeros.

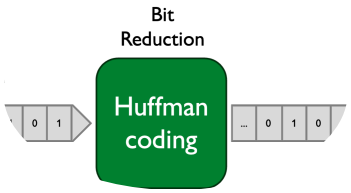
Valid Canon. H. Code

A	110	3
B	10	2
C	0	1
D	111	3

Resources: https://github.com

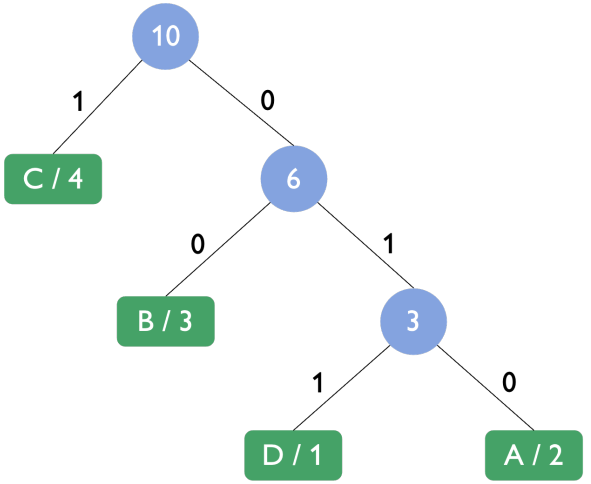
Huffman Coding

CHC: Construction / Deconstruction



Construction Algorithm

1. Create a generic Huffman codebook.

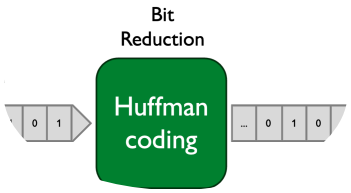


Codes

A	010
B	00
C	1
D	011

Huffman Coding

CHC: Construction / Deconstruction



Construction Algorithm

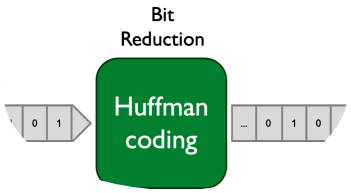
1. Create a generic Huffman codebook.
2. Sort the codes by codeword length.

Codes	
A	010
B	00
C	1
D	011

Codes	
C	1
B	00
A	010
D	011

Huffman Coding

CHC: Construction / Deconstruction



Construction Algorithm

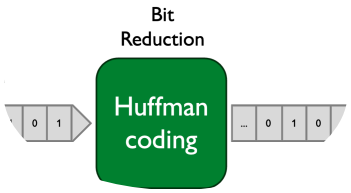
1. Create a generic Huffman codebook.
2. Sort the codes by codeword length.
3. For each length, sort alphabetically by the symbol.

Codes

C	1
B	00
A	010
D	011

Huffman Coding

CHC: Construction / Deconstruction



Construction Algorithm

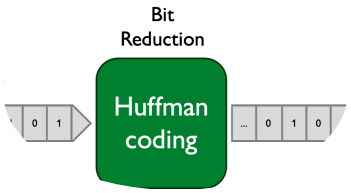
1. Create a generic Huffman codebook.
2. Sort the codes by codeword length.
3. For each length, sort alphabetically by the symbol.
4. The first codeword:
    1. Length * ”0”

Codes

Symbol	Gen. Code	Length	Can. Code
C	1	1	0
B	00	2	
A	010	3	
D	011	3	

Huffman Coding

CHC: Construction / Deconstruction



Construction Algorithm

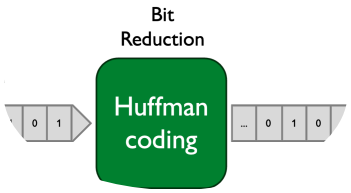
1. Create a generic Huffman codebook.
2. Sort the codes by codeword length.
3. For each length, sort alphabetically by the symbol.
4. The first codeword:
    1. Length * ”0”
5. Each next codeword:
    1. Add ”1” to previous codeword
    2. If len(curr-codeword) > len(prev-codeword), append “0”-padding until len(codeword) == length

Codes

Symbol	Gen. Code	Length	Can. Code
C	1	1	0
B	00	2	10
A	010	3	110
D	011	3	111

Huffman Coding

CHC: Construction / Deconstruction



Codes

Symbol	Gen. Code	Length	Can. Code
C	1	1	0
B	00	2	10
A	010	3	110
D	011	3	111

Can. Codes

Symbol	Length	Can. Code
A	3	110
B	2	10
C	1	0
D	3	111

[3, 2, 1, 3]

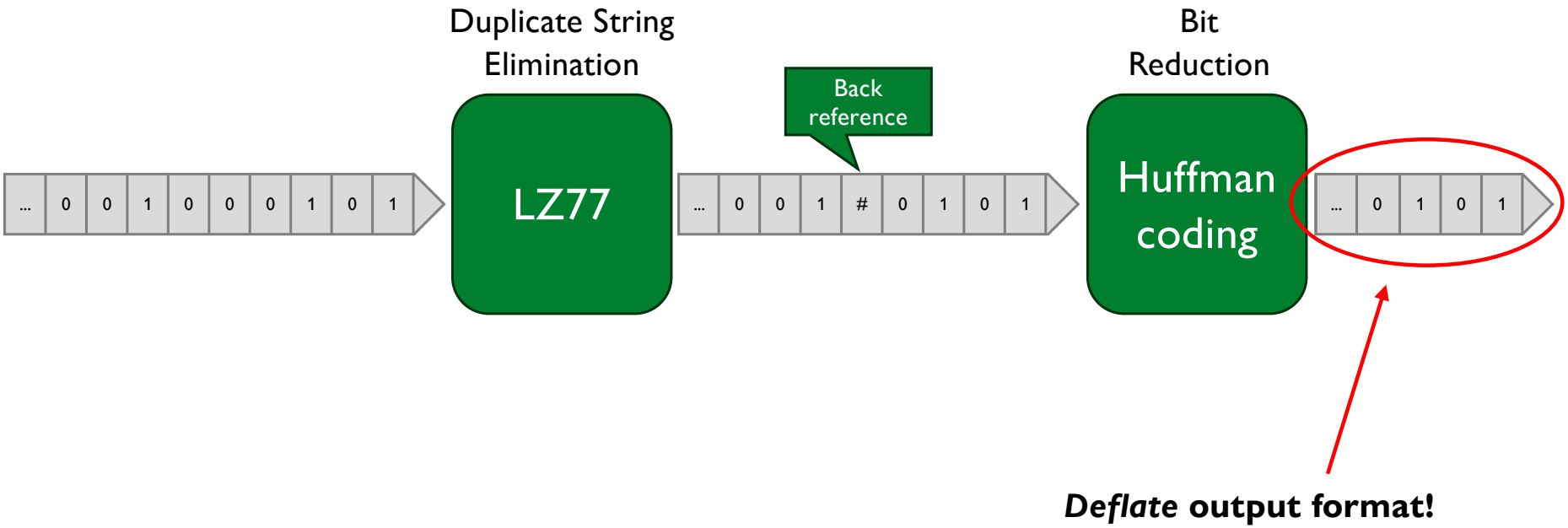
Rules

- The order of the code lengths are representing the alphabetically ordered symbols.
- Code length "0" represents a non-occuring symbol.
- The code words of a given length are ascending ordered.
- The shortest code word consists of only zeros.

Resources: https://github.com

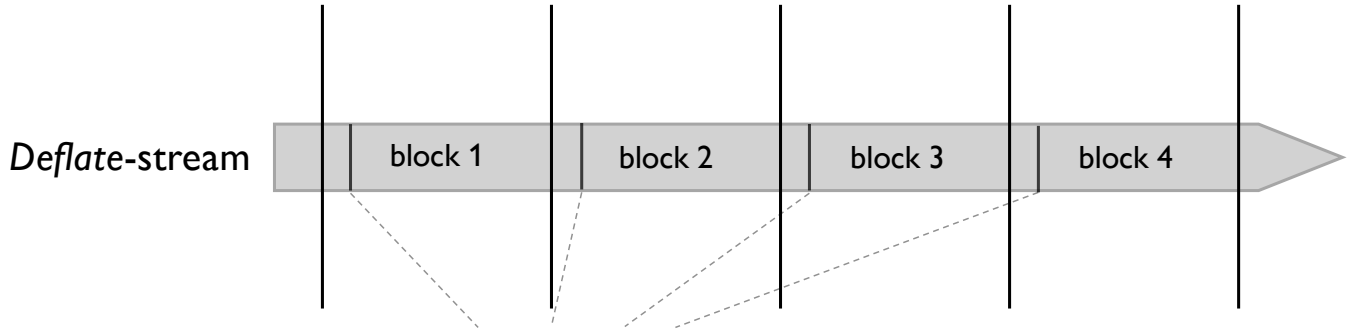
Deflate

In-depth view



Deflate

Basic element – Blocks!



Block Header

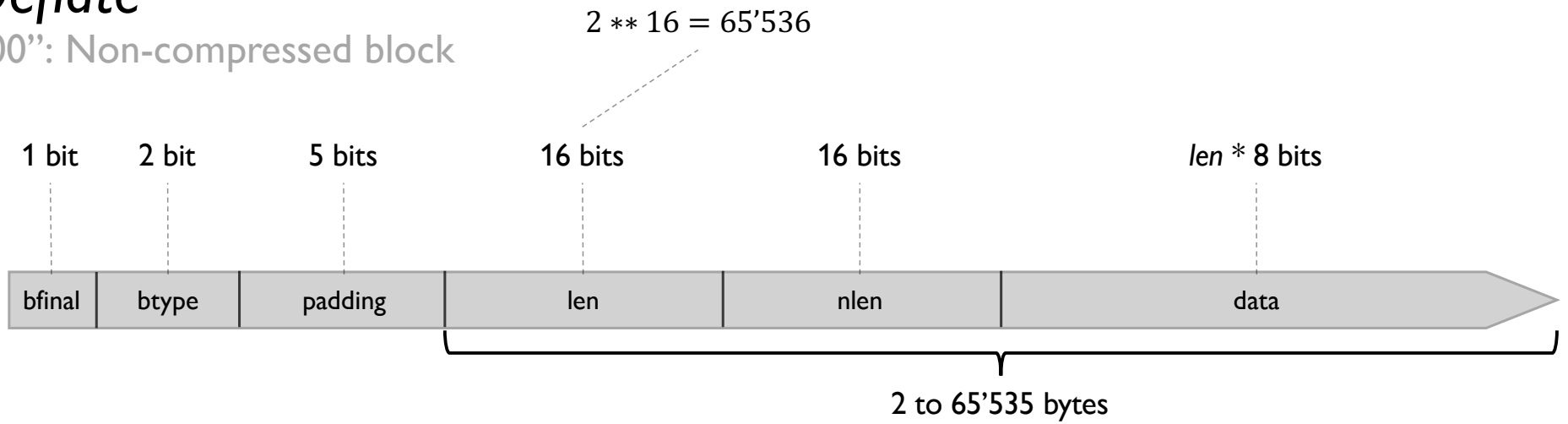
- Bit 1: bfinal
- Bit 2-3: btype

Block Types

Btype	Name	Description	Length
00	Non-compressed block	An non-compressed block, containing literal values.	Between 0 – 65'535 bytes.
01	Static Huffman compressed block	A static Huffman compressed block, using an a priori known Huffman tree which is defined in RFC1951.	Unlimited.
10	Dynamic Huffman compressed block	A dynamic Huffman compressed block, using block-individual transferred Huffman trees.	Unlimited.
11	Reserved block	A reserved block for errors, which should not be used.	n.a.

Deflate

”00”: Non-compressed block



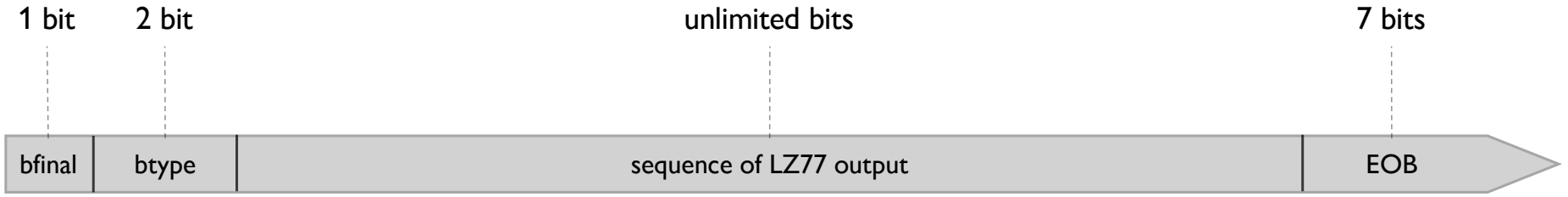
“00”: Non-compressed block structure

- Padding: The block needs to start at the next byte boundary.
- Len: Indicates the number of bytes in the block.
- Nlen: “one’s complement” of “len”; acting as a checksum.
- Data: “len” – 4 bytes of uncompressed data.

Deflate

”01”: Static Huffman comp. block

“LZ77-compressed data encoded with fixed Huffman codes”



Static Huffman codebooks:

- Litlen (little endian) codebook: Literals, pointer lengths, EOB marker
- Dist codebook: Pointer distances

Deflate

”01”: Static Huffman comp. block – Litlen codebook

“LZ77-compressed data encoded with fixed Huffman codes”

Litlen value	Codeword length	Description	Pointer Length	Extra Bits
0-143	8	Literal bytes	-	-
144-255	9	Literal bytes	-	-
256	7	EOB marker	-	-
257-279	7	Pointer length	see lookup table on right	see lookup table on right
280-285	8	Pointer length	see lookup table on right	see lookup table on right
286-287	8	Not used		

Litlen	Extra Bits	Length(s)
257	0	3
258	0	4
259	0	5
260	0	6
261	0	7
262	0	8
263	0	9
264	0	10
265	1	11–12
266	1	13–14
267	1	15–16
268	1	17–18
269	2	19–22
270	2	23–26
271	2	27–30
272	2	31–34
273	3	35–42
274	3	43–50
275	3	51–58
276	3	59–66
277	4	67–82
278	4	83–98
279	4	99–114
280	4	115–130
281	5	131–162
282	5	163–194
283	5	195–226
284	5	227–257
285	0	258

Deflate

”01”: Static Huffman comp. block – Dist codebook

“LZ77-compressed data encoded with fixed Huffman codes”

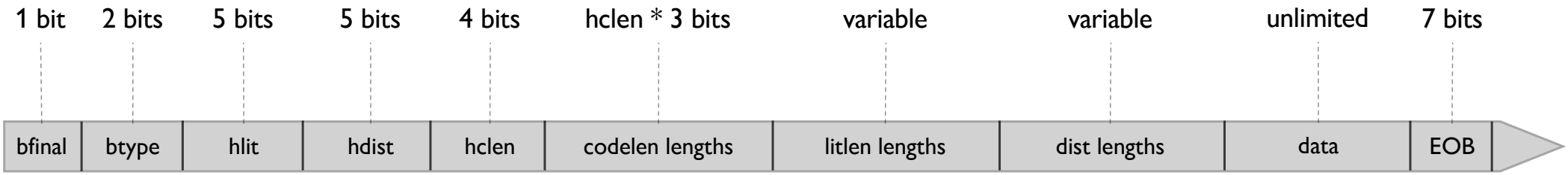
Dist value	Codeword length	Description	Pointer Distance	Extra Bits
0-29	5	Pointer distance	see lookup table on right	see lookup table on right

Dist	Extra Bits	Distance(s)
0	0	1
1	0	2
2	0	3
3	0	4
4	1	5–6
5	1	7–8
6	2	9–12
7	2	13–16
8	3	17–24
9	3	25–32
10	4	33–48
11	4	49–64
12	5	65–96
13	5	97–128
14	6	129–192
15	6	193–256
16	7	257–384
17	7	385–512
18	8	513–768
19	8	769–1024
20	9	1025–1536
21	9	1537–2048
22	10	2049–3072
23	10	3073–4096
24	11	4097–6144
25	11	6145–8192
26	12	8193–12288
27	12	12289–16384
28	13	16385–24576
29	13	24577–32768

Deflate

”10”: Dynamic Huffman comp. block

“LZ77-compressed data encoded with custom Huffman codes”



“10”: Dynamic Huffman comp. block structure

- => New “Codelen” canonical Huffman codebook is used to encode Litlen and Dist codeword lengths.

- Hlit: (5 bits + 257) # of Litlen codeword lengths.
- Hdist: (5 bits + 1) # of Dist codeword lengths.
- Hclen: (4 bits + 4) # of Codelen codeword lengths.
- Codelen lengths: (hclen * 3 bits) 3-bit Codelen lengths read from *deflate* stream in special order.
- Litlen lengths: Litlen length values read from *deflate* stream.
- Dist lengths: Dist length values read from *deflate* stream.
- Data: Literals and pointers (LZ77-compressed data).
- EOB: End-of-block marker.

Deflate

”10”: Dynamic Huffman comp. block – Codelen decoder

“LZ77-compressed data encoded with custom Huffman codes”

Step 1. Get number of length values that should be read from the *deflate*-stream: “hlit”, “hdist”, “hclen”

Step 2. Read “hclen”-amount of “codelen lengths” from the *deflate*-stream, and map them to the alphabet consecutively. Derive codebook!
Those 3-bit values are the codelen codeword lengths.

16, 17, 18, 0, 8, 7, 9, 6, 10, 5,
11, 4, 12, 3, 13, 2, 14, 1, 15

```
0 - 15: Represent code lengths of 0 - 15
    16: Copy the previous code length 3 - 6 times.
        The next 2 bits indicate repeat length
              (0 = 3, ... , 3 = 6)
           Example:  Codes 8, 16 (+2 bits 11),
                     16 (+2 bits 10) will expand to
                     12 code lengths of 8 (1 + 6 + 5)
    17: Repeat a code length of 0 for 3 - 10 times.
        (3 bits of length)
    18: Repeat a code length of 0 for 11 - 138 times
        (7 bits of length)
```

hclen = “0001” => 1 + 4 = 5

codelen lengths = “101|110|111|000|001”

16	17	18	0	8	7	9	...
5	6	7	0	1	0	0	...

Derive codebook!

Deflate

"LZ77-compressed data encoded with custom Huffman codes"

"10": Dynamic Huffman comp. block – Codelen decoder

Step 1. Get number of length values that should be read from the *deflate*-stream: "hlit", "hdist", "hclen"

Step 2. Read "hclen"-amount of "codelen lengths" from the *deflate*-stream, and map them to the alphabet consecutively. Derive codebook!
Those 3-bit values are the codelen codeword lengths.

Step 3. Decode litlen lengths and dist lengths, using codelen lengths codebook, and derive the litlen and dist codebooks!

16	17	18	0	8	7	9	...
5	6	7	0	1	0	0	...

```
0 - 15: Represent code lengths of 0 - 15
    16: Copy the previous code length 3 - 6 times.
        The next 2 bits indicate repeat length
              (0 = 3, ... , 3 = 6)
           Example:  Codes 8, 16 (+2 bits 11),
                     16 (+2 bits 10) will expand to
                     12 code lengths of 8 (1 + 6 + 5)
    17: Repeat a code length of 0 for 3 - 10 times.
        (3 bits of length)
    18: Repeat a code length of 0 for 11 - 138 times
        (7 bits of length)
```

Deflate

"LZ77-compressed data encoded with custom Huffman codes"

"10": Dynamic Huffman comp. block – Codelen decoder

Step 1. Get number of length values that should be read from the *deflate*-stream: "hlit", "hdist", "hclen"

Step 2. Read "hclen"-amount of "codelen lengths" from the *deflate*-stream, and map them to the alphabet consecutively. Derive codebook!
Those 3-bit values are the codelen codeword lengths.

Step 3. Decode litlen lengths and dist lengths, using codelen lengths codebook, and derive the litlen and dist codebooks!

Step 4. Decompress "data" literals and pointers using the derived custom litlen and dist codebooks.

Deflate

"LZ77-compressed data encoded with custom Huffman codes"

"10": Dynamic Huffman comp. block – Codelen decoder

Step 1. Get number of length values that should be read from the *deflate*-stream: "hlit", "hdist", "hclen"

Step 2. Read "hclen"-amount of "codelen lengths" from the *deflate*-stream, and map them to the alphabet consecutively. Derive codebook!
Those 3-bit values are the codelen codeword lengths.

Step 3. Decode litlen lengths and dist lengths, using codelen lengths codebook, and derive the litlen and dist codebooks!

Step 4. Decompress "data" literals and pointers using the derived custom litlen and dist codebooks.

Step 5. When EOB marker is decoded, stop decompression, and continue with next block.

Deflate

Why use different block types?

Short Recap:

- **Non-compressed** blocks are raw data (not LZ77-compressed) consisting of bytes of symbols.
- **Static Huffman** compressed blocks have data that consists of literals and pointers. (No overhead transm. trees).
- **Dynamic Huffman** compressed blocks have data that consists of literals and pointers. (Notable overhead transmitting custom trees).

"***Deflate*** is only a **format specification**."

"**Smart decisions** about **when** to compress in **which** block type."

When to use which block type?

Already highly compressed data (e.g. image in JPEG).

When processing speed is more important than transmission speed.

Data that contains both repetitive, and non-repetitive sections of data (e.g. scientific paper).

When data is repetitive, but the patterns do not change much over time (e.g. log files).

When dealing with very short pieces of data.

...

Thank you!

Länzlinger Jonas
M.Sc. Candidate in Computer Science
+41 (0)77 479 15 42
jonas.laenzlinger@student.unisg.ch

University of St.Gallen

University of St.Gallen
Dufourstrasse 50
9000 St.Gallen

unisg.ch

Accreditation

EQUIS ACCREDITED
AACSB ACCREDITED
ASSOCIATION OF MBAS ACCREDITED