CS 444 Operating Systems

Implementation of Huffman Code

September 6, 2022

Apply for a CS Account

- Apply for an account on the CS subnet within UMB here `https://portal.cs.umb.edu/accounts/login/`
- Enter a username of your choice
- Enter your UMB email — you will receive a link to activate your CS account
- Enter a password of your choice
- Join CS 444
- See step-by-step directions here `https://www.cs.umb.edu/~ghoffman/linux/apply_process.html`

Text File Compression

- Huffman code
- David Huffman, 1952

Example

- Compress a file of 100,000 characters of a, b, c, d, e, and f
- Fixed-length codes: 300,000 bits
- Variable-length codes:

	a	b	c	d	e	f
Frequency (in 1000s)	45	13	12	16	9	5
Fixed-length codes	000	001	010	011	100	101
Variable-length codes	0	101	100	111	1101	1100

Lossless File Compression

- File compression using reduced representation of characters
- Let F be a file with n characters (n bytes or $8n$ bits)
- Each byte is a binary representation of the ASCII code of a character
- Represent every character using a unique *code* of m bits ($m < 8$), and write a file F' with the original characters replaced by their codes
- The new file size is $8m < 8n$ bits
- *Lossless* compression
  - We should be able to decompress F' and get F back

Rationale

- Not all of 8 bits are needed to uniquely represent a character
- Most files contain much fewer than 256 different characters
- Consider a file of decimal digits and blanks and newlines
  - 12 characters
- As an ASCII file, each `char` takes 8 bits
- But we can use 4 bits to code for 12 characters
  - $0x30 \rightarrow 0$: digit '0' becomes 0000_2
  - $0x31 \rightarrow 1$
  - ...
  - $0x39 \rightarrow 9$
  - $0x0A \rightarrow 0xA$: linefeed (LF)
  - $0x20 \rightarrow 0xB$: space ' ' becomes 1011_2
- Such a file can be compressed by a factor of 2

Example

Char	ASCII (dec)	ASCII (hex)	ASCII (binary)	New Code
'0'	48	30	00110000	0000
'1'	49	31	00110001	0001
'2'	50	32	00110010	0010
'3'	51	33	00110011	0011
'4'	52	34	00110100	0100
'5'	53	35	00110101	0101
'6'	54	36	00110110	0110
'7'	55	37	00110111	0111
'8'	56	38	00111000	1000
'9'	57	39	00111001	1001
LF	10	0A	00001010	1010
SP	32	20	00100000	1011

Compress and Decompress

- Original file "00123 890\n00456 098\n"
- In ASCII hex: 30 30 31 32 33 20 38 39 30 0A 30 30 34 35 36 20 30 39 38 0A
- Compressed file in hex: 00 12 3B 89 0A 00 45 6B 09 8A (half as long)
- To decompress, read 4 bits at a time from the compressed file
  - $0 \rightarrow 0x30, 1 \rightarrow 0x31, \ldots, 0xA \rightarrow 0x0A, 0xB \rightarrow 0x20$
- Hex is easier to work with in this case
- Question: When decompressing a file, how do you know you have reached the end of the file?
  - The last byte may contain only 1 nibble of code, or 2 nibbles

Efficient File Compression

- We would like a quantitative estimate of the effectiveness of various coding schemes
- We need a distribution of *character frequencies*
- Consider this distribution
  - Digits 1 through 9 are about equally likely, although of declining frequencies
  - 0, space, and linefeed are much more frequent

Character Frequency

Char	SP	LF	0	1	2	3	4	5	6	7	8	9
Freq	30	20	10	7	6	5	4	3	3	3	2	2

Total frequency count is 95

- With this distribution, we would like shorter codes for SP, LF, and 0, and longer ones for the other digits
- How do we decompress if the lengths of the codes are variable?
- The answer is *prefix codes*

Prefix Codes

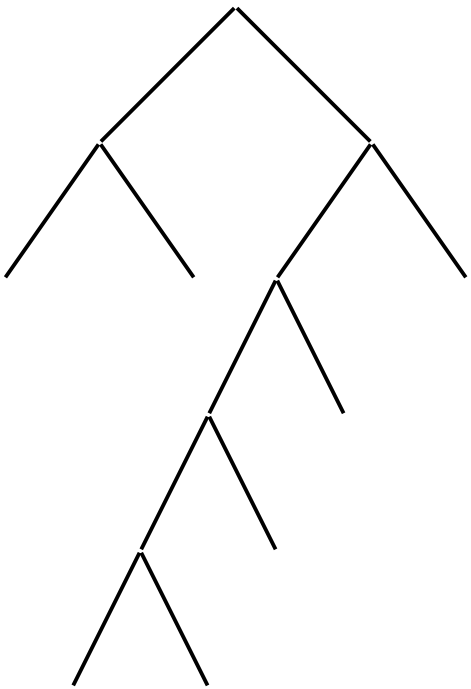
- Prefix means some initial substring
- For example 110 is a prefix of 11011
- A set of prefix codes has the property that no code is the prefix of another
- With a set of prefix codes, if we match the initial bits (prefix) of the compressed data with all the bits of a certain code, it can only be that code
- So we look no further, remove the prefix from the stream, decode it, and repeat

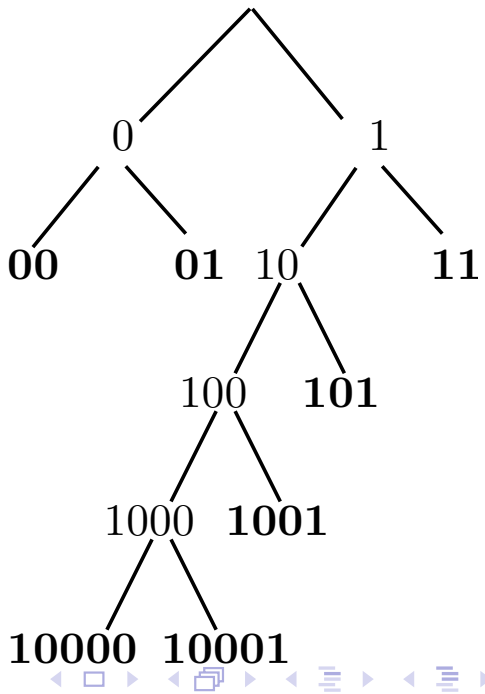
Prefix Code Examples

- For example, { 00, 10, 110 } is a set of prefix codes, because all three pass the test:
  - Testing 00: neither 10 nor 110 start with 00
  - Testing 10: neither 00 nor 110 start with 10
  - Testing 110: neither 00 nor 10 start with 110
- { 0, 10, 11 } is also a set of prefix codes
- { 0, 01, 11 } is not a set of prefix codes because 0 is a prefix of 01

Construct a Prefix Code Tree

- How do we generate a set of prefix codes for a certain use?
- Answer: Construct a binary tree with the right number of leaves
- Each code is determined by a *path* from the root to the leaf, where going left gives a 0 and going right a 1
- For example:





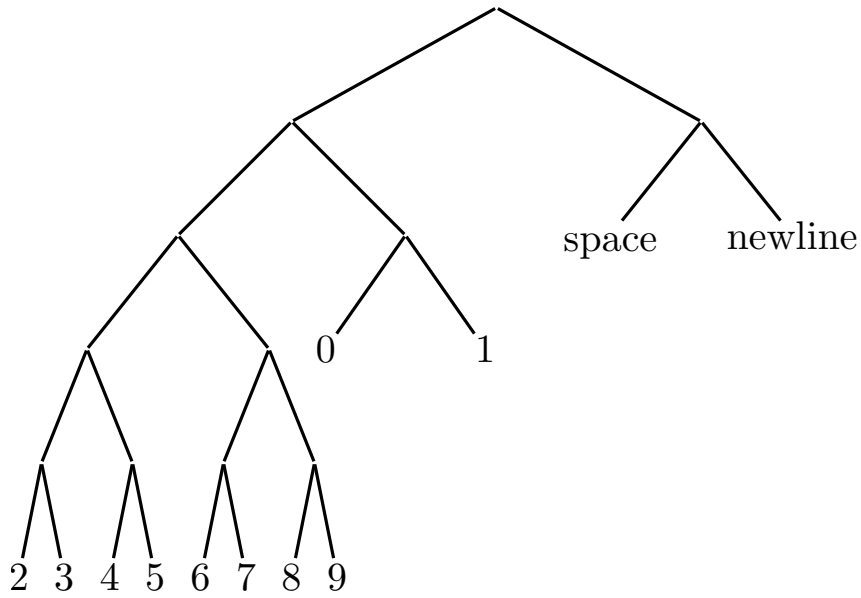
Read the Prefix Codes from a Code Tree

- This defines
  - 00 and 01 from the leaves at the left
  - 10000 and 10001 for the leaves at the bottom
  - 10001, 1001, 101, and 11 for the leaves going up the right-hand side
- Now we have some short codes for frequent symbols, and some longer codes for less frequent symbols
- No bit string is a prefix of another, because each bit string specifies a unique path to a leaf

Assign Codes According to Frequency

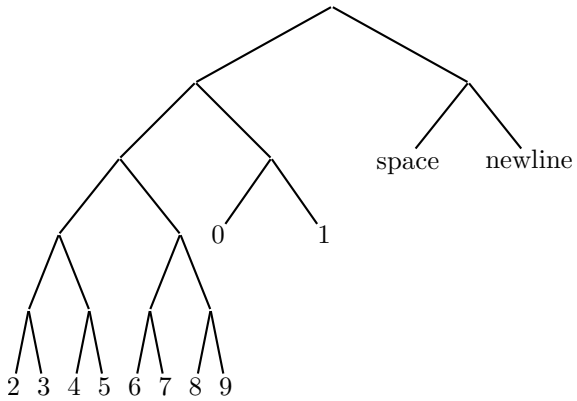
Char	SP	LF	0	1	2	3	4	5	6	7	8	9
Freq	30	20	10	7	6	5	4	3	3	3	2	2

- Our example of 12 symbols
  - Digits 1 through 9 are about equally likely, although of declining frequencies (this is actually observed)
  - 0, space, and linefeed are much more frequent
- We can set up a binary tree with these 12 symbols at the leaves



Convert Symbols to Codes

- Read off the codes from the binary tree
- Linefeed is reached by traversing down the right-hand side of the tree, going right 2 times, so its code is 11
- Space is reached by going right and then left, so its code is 10
- 1 is reached by going left, then right, then right, so its code is 011, and so on
- 291 bits is used to encode the file, much better than $4 \times 95 = 380$ bits



char	code	freq	total bits
SP	10	30	60
LF	11	20	40
0	010	10	30
1	011	7	21
2	00000	6	30
3	00001	5	25
4	00010	4	20
5	00011	3	15
6	00100	3	15
7	00101	3	15
8	00110	2	10
9	00101	2	10

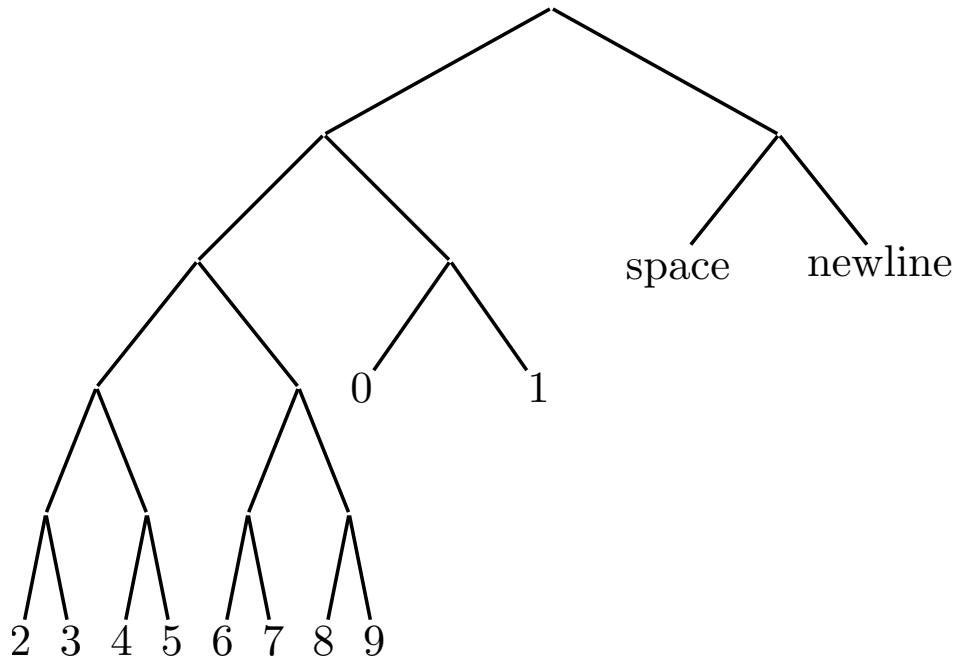
Encode the File

- Original file "00123890\$n$00456098\$n$"
- In hex: 30 30 31 32 33 20 38 39 30 0A 30 30 34 35 36 20 30 39 38 0A
- Compressed file in binary: 010 010 011 00000 00001 10 00110 00111 010 11 010 010 00010 00011 00100 ...
- Question: When compressing a file, what do you do with the last byte, which may not be completely filled with codes?

Decompress a File

- For example: 010010001001010001111 1
- Following the bits from the input file, traverse down the code tree until a leaf is reached
- 010 leads to the leaf labeled by "0"

010	$\rightarrow$	0
00100	$\rightarrow$	6
10	$\rightarrow$	sp
10	$\rightarrow$	sp
00111	$\rightarrow$	9
11	$\rightarrow$	nl



- Thus this bit string decodes to "006 9\n"
- Is this an optimal coding?

Huffman Coding

Char	sp	nl	0	1	2	3	4	5	6	7	8	9
Freq.	30	20	10	7	6	5	4	3	3	3	2	2

4

Char	sp	nl	0	1	2	3	8	9	4	5	6	7
Freq.	30	20	10	7	6	5	2	2	4	3	3	3

4 6

Char	sp	nl	0	1	6	7	2	3	8	9	4	5
Freq.	30	20	10	7	3	3	6	5	2	2	4	3

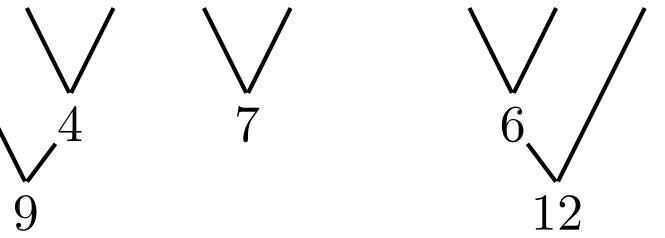
6 4 7

Char	sp	nl	0	4	5	1	6	7	2	3	8	9
Freq.	30	20	10	4	3	7	3	3	6	5	2	2

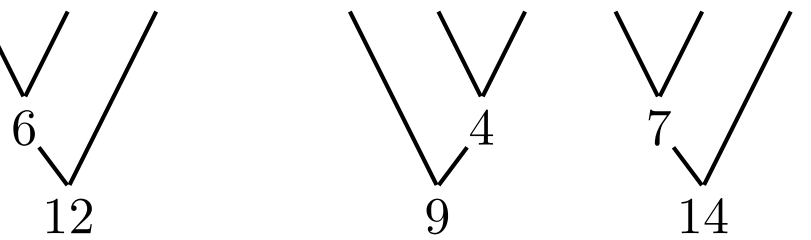
7 6 4 9

Huffman Coding

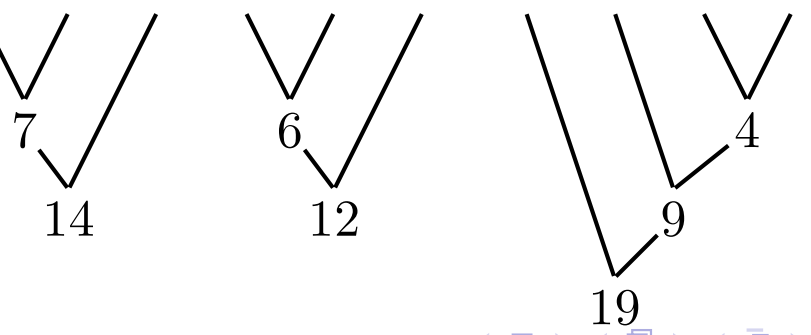
Char	sp	nl	0	3	8	9	4	5	1	6	7	2
Freq.	30	20	10	5	2	2	4	3	7	3	3	6



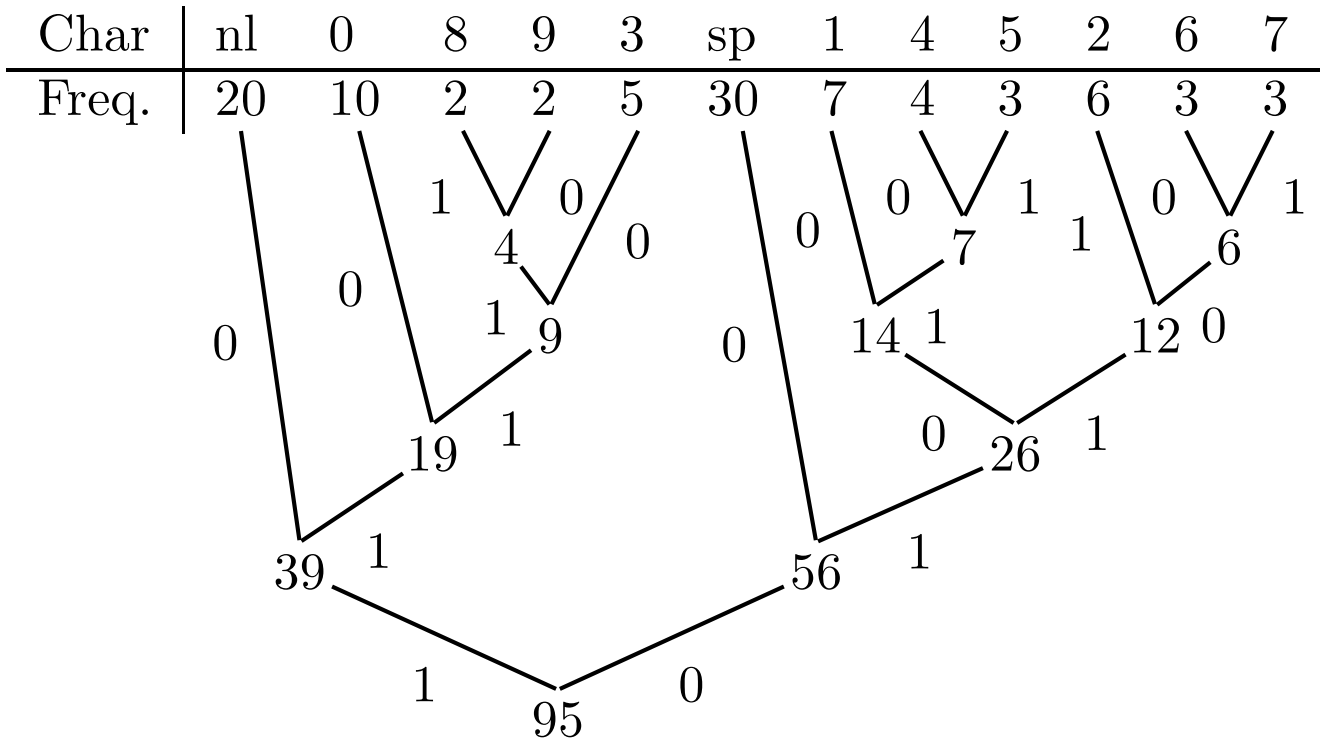
Char	sp	nl	6	7	2	0	3	8	9	4	5	1
Freq.	30	20	3	3	6	10	5	2	2	4	3	7



Char	sp	nl	4	5	1	6	7	2	0	3	8	9
Freq.	30	20	4	3	7	3	3	6	10	5	2	2



Huffman Coding

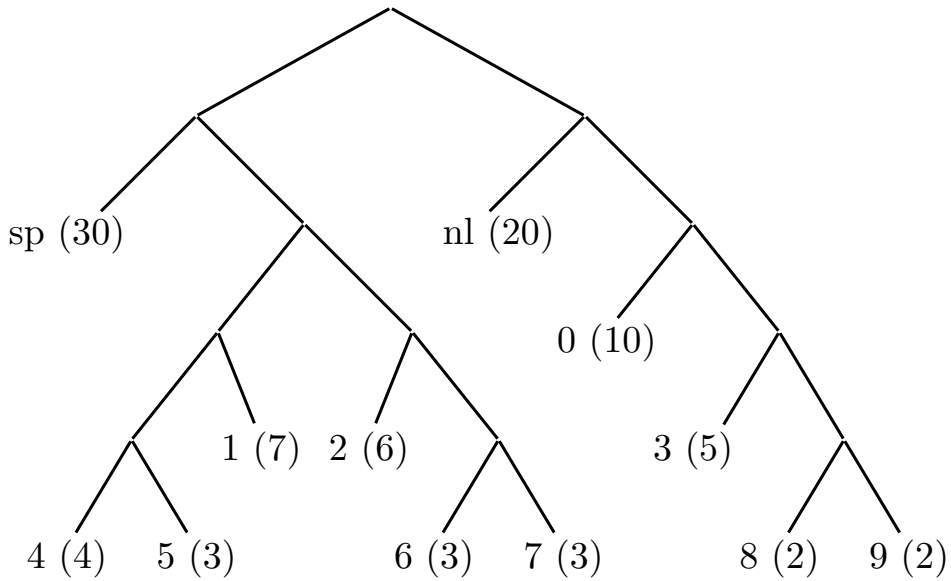


- During tree construction, a *priority queue* holds the characters
- The frequencies of the characters are their priorities
- Convention to assign code: larger weight = 0, smaller weight = 1, random code for same weights

Huffman Coding

char	code	freq.	total bits
SP	00	30	60
LF	10	20	40
0	110	10	30
1	0101	7	28
2	0110	6	24
3	1110	5	20
4	01000	4	20
5	01001	3	15
6	01110	3	15
7	01111	3	15
8	11110	2	10
9	11111	2	10

Total bits = 287



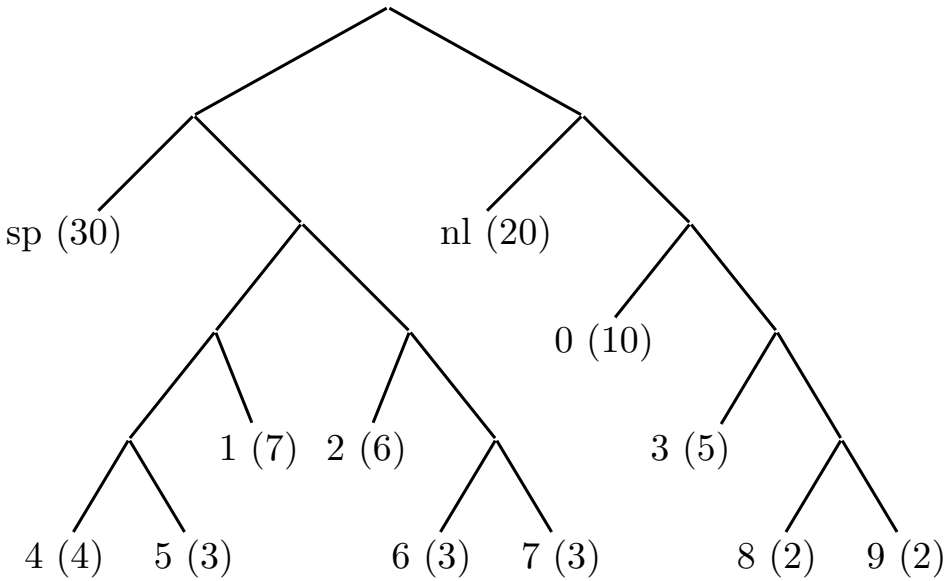
Why is Huffman's Algorithm Optimal?

- Consider the optimal coding scheme for n characters
- It will have a longest code for the least frequent symbol
- It will have at least two codes of this longest length, or we could shorten one
  - In other words, there are no nodes with one child, or we can replace the child by the parent and get a shorter tree
- The two symbols of the least frequencies will be of this particular longest length
- Thus an optimal coding scheme has its two least frequent symbols with codes of the same length, the longest code length
  - This is the first step of constructing the Huffman code tree
  - The base case of induction

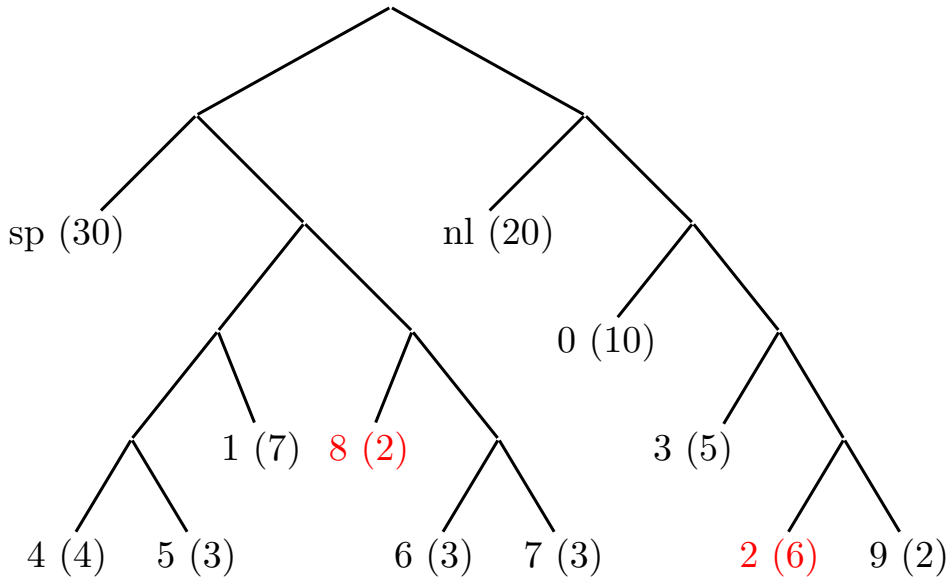
Why is Huffman's Algorithm Optimal?

- If there are more than two codes of this length, they are interchangeable, so get the two least frequent ones paired up to share all but the last bit of their codes
- Then merge the two characters into one new imaginary symbol with summed frequency
- The optimal coding scheme for this new char set will yield the optimal coding scheme for the original set
  - The induction
- In this way, we keep coalescing until the problem is trivial, when we have a rooted binary tree

Why is Huffman's Algorithm Optimal?

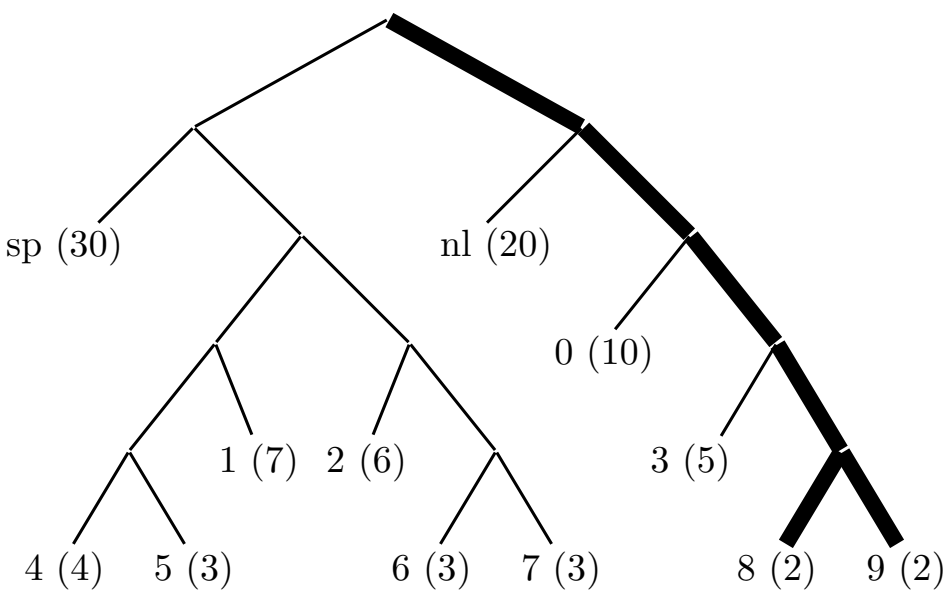


Huffman's tree

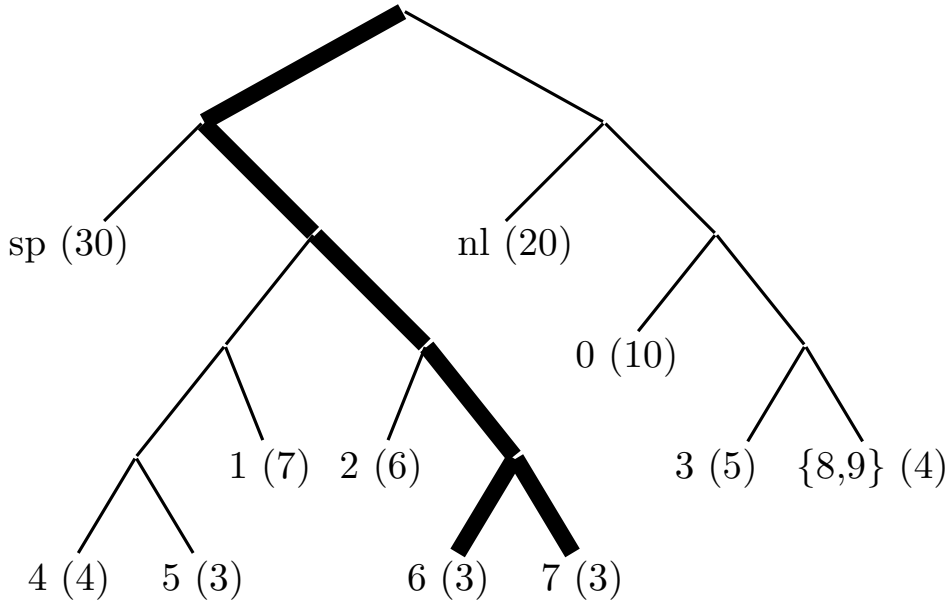


A slightly different tree, less effective in compression than Huffman's

Why is Huffman's Algorithm Optimal?



Longest path before compression



Longest path after compression

Implementation of Huffman's Algorithm

- Generate a custom table from the character frequencies of the document under consideration
- Count character frequencies of the file, build a Huffman tree, then compress its contents using the Huffman codes
- Send the frequency table and the compressed document to the recipient, where the same Huffman tree can be built and used to decompress
- Implementation
  1. Read the file once to collect character frequencies
  2. Build a Huffman tree based on the character frequencies
  3. Read the codes from the tree
  4. Read the file again and, for each ASCII code, output its Huffman code

Bitwise Operations

- & bitwise AND
- | bitwise inclusive OR
- ^ bitwise exclusive OR
- << left shift
- >> right shift
- ~ one's complement
- – two's complement

One's Complement

- Unary operator: ~
- Flip each bit in the operand
- Zeros become ones
- Ones become zeros
- Example: `~10101010 == 01010101`

Two's Complement

- Unary operator: -
- Two's complement is the negation operation
- It performs the following:

1. Take the one's complement of the operand
2. Then add 1

- Try two's complement for these: `0x55`, `0`, -2^7

Left Shift <<

- Syntax: operand << numOfBitPositions
- Shift the bits in operand to the left
- Bits that fall off the left side will disappear
- 0's are shifted in from the right
- The operand is usually an unsigned integer
- The number of bit positions must be *positive*
- The value of `i << 0` is not defined
- Example: `0x55 << 3`
- Left-shifting is the same as multiplying by a power of 2

Right Shift >>

- Syntax: operand >> numOfBitPositions
- Shift the bits in operand to the right
- Bits that fall off the right side will disappear
- If the operand is unsigned, 0's are shifted in from the left
- If the operand is signed
  - Arithmetic shift: fills with sign bit (extension)
  - Logical shift: fills with 0's
- The number of bit positions must be *positive*
- The value of `i >> 0` is not defined
- Example: `0x55 >> 1`
- Right-shifting of unsigned is the same as dividing by a power of 2

Bitwise AND, Inclusive OR, Exclusive OR

- Take the logical AND, OR, and XOR
- Apply to each pair of bits

```
  01001000       01001000       01001000
& 10111000     | 10111000     ^ 10111000
----------     ----------     ----------
  00001000       11111000       11110000
```

- Example
  - `x = x & ~077` will clear the lowest 6 bits of `x`
  - `q = x >> 077` is the quotient of dividing by 64
  - `r = x & 077` is the remainder

Bit Masking

- The process of turning on or off some bits in specific positions in an unsigned integer
- Some programs require a large number of Boolean variables
- These variables are often referred to as flags
- Since C does not have a Boolean type, these flags require an integer type variable, but this uses more memory than is necessary for a variable that only needs the capacity to hold two values
- To save memory, these types of variables are often packed into one integer variable

Masks to Retrieve One Bit, Method 1

```
#define FLAG_1   1 /* 0000 0001 */
#define FLAG_2   2 /* 0000 0010 */
#define FLAG_3   4 /* 0000 0100 */
#define FLAG_4   8 /* 0000 1000 */
#define FLAG_5  16 /* 0001 0000 */
#define FLAG_6  32 /* 0010 0000 */
#define FLAG_7  64 /* 0100 0000 */
#define FLAG_8 128 /* 1000 0000 */
void main(void) {
  int state = 44; /* 0010 1100 */
  if (state & FLAG_1) printf("Flag 1 is set\n");
  if (state & FLAG_2) printf("Flag 2 is set\n");
  if (state & FLAG_3) printf("Flag 3 is set\n");
  if (state & FLAG_4) printf("Flag 4 is set\n");
  if (state & FLAG_5) printf("Flag 5 is set\n");
  if (state & FLAG_6) printf("Flag 6 is set\n");
  if (state & FLAG_7) printf("Flag 7 is set\n");
  if (state & FLAG_8) printf("Flag 8 is set\n");
}
```

Masks to Retrieve One Bit, Method 2

```
#include <stdio.h>
#include <stdint.h>

uint64_t bitPos[64];

int main(void) {
  uint64_t i, state = 0x1234567890ABCDEFul;

  bitPos[0] = 1;
  for (i = 1; i < 64; i++)
    bitPos[i] = bitPos[i - 1] << 1;

  for (i = 0; i < 64; i++)
    if (state & bitPos[i])
      printf("bit %lu is set\n", i + 1);

  return 0;
}
```

Turn Off a Group of Bits

- Example: `char n = '\xA5';`
  `1010 0101`
- Turn off the most significant 3 bits
  Equivalently, keep the least significant 5 bits
- So create a mask `0001 1111`, which is `'\x1F'`
- Do a bitwise AND between `n` and the mask
- `n = n & '\x1F';`

Turn Off a Group of Bits

- Example: `char n = '\xA5';`
  `1010 0101`
- Turn off the least significant 6 bits
  Equivalently, keep the most significant 2 bits
- So create a mask `1100 0000`, which is `'\xC0'`
- Do a bitwise AND between `n` and the mask
- `n = n & '\xC0';`
- Alternatively, flip `0011 1111`
  `n = n & ~077`
- `~077` is probably easier than `'\xC0'` for most people

Turn On a Group of Bits

- Example: `char n = '\xA5';`
- `1010 0101`
- Turn on the most significant 2 bits
- `n = n | ~077`

Retrieve a Group of Bits

- Goal: given an `unsigned int x`, retrieve n bits starting at position p
- Example: `n == 3`, `p == 6`, retrieving bits at 6, 5, and 4

7	6	5	4	3	2	1	0

```
unsigned getBits(unsigned x, unsigned p, unsigned n) {
  return (x >> (p - n + 1)) & ~(~0 << n);
}
```

Big Endian and Little Endian

- Big endian: the 4 bytes of `int` is stored from the most significant byte to the least significant
- Little endian: reversed
- We can test to see whether a machine is big or little endian

```
union {
  int i;
  char c[sizeof(int)];
} u;

u.i = 1;
if (u.c[0] == 1)
  printf("little endian\n");
else
  printf("big endian\n");
```

Binary File I/O

- When we open files in `"r"`, `"w"`, or `"a"` modes, they are text files
- Data for I/O are converted to ASCII codes
- We can write binary data directly to files
- Binary file I/O is faster than ASCII I/O
- Binary files are more compact
- Binary files preserve all bits — text files incur loss of precision

fread() and fwrite()

```
size_t fread(void *ptr, size_t size,
             size_t howMany, FILE *fp);

size_t fwrite(const void *ptr, size_t size,
              size_t howMany, FILE *fp);
```

Write a Binary File, `fwrite()`

- We can use `"rb"`, `"wb"`, or `"ab"` modes to open binary files

```
uint32_t i;
FILE *fp;
fp = fopen("binaryFile", "wb");
if (!fp) {
  fprintf(stderr, "fail to open file\n");
  return 1;
}
for (i = 0; i < 20; i++)
  fwrite(&i, sizeof(uint32_t), 1, fp);
fclose(fp);
```

Read a Binary File, fread()

```
fp = fopen("binaryFile", "rb");
if (!fp) {
  fprintf(stderr, "fail to open file\n");
  return 1;
}
fread(num, sizeof(uint32_t), 20, fp);
fclose(fp);

for (i = 0; i < 20; i++)
  printf("%u\t%u\n", i, num[i]);
```

Save Data to a Binary File

- Java: serialization
- Python: pickle
- C: write a large data structure to a binary file
- Be careful with endianness
- You should fully document the format and how to read/write to ensure portability

How to View Binary Files

- Use the command `hexdump` to examine a binary file
- Do a `hexdump` of a small executable
- ELF, executable and linkable format
- Use the command `locate` to locate a file
- Use the utility `readelf` to see what is in an ELF file, such as `libc.so`