CSC 225 - Summer 2019

Compression and Huffman Coding

Bill Bird

Department of Computer Science
University of Victoria

June 19, 2019

Compression (1)

```
It was the best of times, it was the worst of times, it
was the age of wisdom, it was the age of foolishness, it
was the epoch of belief, it was the epoch of incredulity,
it was the season of Light, it was the season of Darkness,
it was the spring of hope, it was the winter of despair,
we had everything before us, we had nothing before us, we
were all going direct to Heaven, we were all going direct
the other way
```

416 characters, 28 distinct characters

Consider the text above, which is an excerpt from *A Tale of Two Cities* by Charles Dickens.

Compression (2)

```
It was the best of times, it was the worst of times, it
was the age of wisdom, it was the age of foolishness, it
was the epoch of belief, it was the epoch of incredulity,
it was the season of Light, it was the season of Darkness,
it was the spring of hope, it was the winter of despair,
we had everything before us, we had nothing before us, we
were all going direct to Heaven, we were all going direct
the other way
```

416 characters, 28 distinct characters

This excerpt, if stored as plain text, requires 416 bytes (or 3328 bits) if each character is stored as one byte (e.g. with an ASCII encoding). One byte is 8 bits, so 256 distinct characters can be encoded as single bytes.

Compression (3)

```
It was the best of times, it was the worst of times, it
was the age of wisdom, it was the age of foolishness, it
was the epoch of belief, it was the epoch of incredulity,
it was the season of Light, it was the season of Darkness,
it was the spring of hope, it was the winter of despair,
we had everything before us, we had nothing before us, we
were all going direct to Heaven, we were all going direct
the other way
```

416 characters, 28 distinct characters

However, the text only contains 28 distinct characters (including spaces and punctuation), so using one byte for each character is unnecessary.

Compression (4)

```
It was the best of times, it was the worst of times, it
was the age of wisdom, it was the age of foolishness, it
was the epoch of belief, it was the epoch of incredulity,
it was the season of Light, it was the season of Darkness,
it was the spring of hope, it was the winter of despair,
we had everything before us, we had nothing before us, we
were all going direct to Heaven, we were all going direct
the other way
```

416 characters, 28 distinct characters

If each character is encoded using only 5 bits, only 2080 bits (or 260 bytes) are necessary to store the entire excerpt. Since $2^5 = 32$, it is possible to represent 28 distinct characters in a 5 bit encoding.

Compression (5)

```
It was the best of times, it was the worst of times, it
was the age of wisdom, it was the age of foolishness, it
was the epoch of belief, it was the epoch of incredulity,
it was the season of Light, it was the season of Darkness,
it was the spring of hope, it was the winter of despair,
we had everything before us, we had nothing before us, we
were all going direct to Heaven, we were all going direct
the other way
```

416 characters, 28 distinct characters

If we encode the text with 5 bits per character, we can easily implement a decoding algorithm that recovers the original plain text representation by reversing the encoding process.

Compression (6)

```
It was the best of times, it was the worst of times, it
was the age of wisdom, it was the age of foolishness, it
was the epoch of belief, it was the epoch of incredulity,
it was the season of Light, it was the season of Darkness,
it was the spring of hope, it was the winter of despair,
we had everything before us, we had nothing before us, we
were all going direct to Heaven, we were all going direct
the other way
```

416 characters, 28 distinct characters

Encoding the text with 5 bits instead of 8 bits is a simple example of a **compression scheme**. The goal of a compression scheme is to reduce the amount of space needed to store a piece of data.

Compression (7)

```
It was the best of times, it was the worst of times, it
was the age of wisdom, it was the age of foolishness, it
was the epoch of belief, it was the epoch of incredulity,
it was the season of Light, it was the season of Darkness,
it was the spring of hope, it was the winter of despair,
we had everything before us, we had nothing before us, we
were all going direct to Heaven, we were all going direct
the other way
```

416 characters, 28 distinct characters

In this course, we will only consider **lossless** compression schemes. A lossless compression scheme compresses a block of data D to a compressed representation C such that C can be decompressed to give exactly the same data D that was originally compressed.

Compression (8)

```
It was the best of times, it was the worst of times, it
was the age of wisdom, it was the age of foolishness, it
was the epoch of belief, it was the epoch of incredulity,
it was the season of Light, it was the season of Darkness,
it was the spring of hope, it was the winter of despair,
we had everything before us, we had nothing before us, we
were all going direct to Heaven, we were all going direct
the other way
```

416 characters, 28 distinct characters

In a lossless scheme, no differences are allowed between the decompressed data and the original source data.

Compression (9)

The alternative is **lossy compression**, where some information is deliberately discarded to save space. JPEG compression (used for images) is an example of lossy compression.

Compression (10)

Many lossy compression methods, including JPEG, use lossless techniques like Huffman Coding, combined with other transformations.

Compression (11)

Lossy compression introduces an extra variable: a **quality** setting. Lower quality levels will sacrifice more detail but save more space, while higher quality levels will save detail and require more space. (The image on the right above has a very low quality setting).

Compression (12)

Image compression is often covered by CSC 205.

Compression (13)

Algorithm	**Size (bytes)**
Original Data	416
5-bit encoding	260
`DEFLATE` (Zip)	175
`gzip`	187
BZip2	227
LZMA (7zip)	252

Popular general purpose compression algorithms include `DEFLATE` (used in zip files), `gzip`, `bzip2` and `LZMA` (used for the 7-zip format).

Compression (14)

Algorithm	**Size (bytes)**
Original Data	416
5-bit encoding	260
`DEFLATE` (Zip)	175
`gzip`	187
BZip2	227
LZMA (7zip)	252

The table above shows the size of the compressed data resulting from applying each algorithm to the text example on the previous slides.

Compression (15)

Algorithm	Size (bytes)
Original Data	753184
7-bit encoding	659036
`DEFLATE` (Zip)	280484
`gzip`	280496
BZip2	203269
LZMA (7zip)	230944

The table above shows the size of the compressed data resulting from encoding the entire text of `A Tale of Two Cities`. Note that the 5-bit scheme can't be applied to the full text, since there are 73 distinct characters in the full book.

Compression (16)

Algorithm	**Size (bytes)**
Original Data	753184
7-bit encoding	659036
`DEFLATE` (Zip)	280484
`gzip`	280496
BZip2	203269
LZMA (7zip)	230944

Since $2^6 < 73 \leq 2^7 = 128$, we can use a 7-bit encoding, but relatively little compression is achieved.

Compression (17)

Algorithm	**Size (bytes)**
Original Data	753184
7-bit encoding	659036
`DEFLATE` (Zip)	280484
`gzip`	280496
BZip2	203269
LZMA (7zip)	230944

We want to develop a general purpose compression algorithm, which gives decent performance even when the input data is arbitrary binary data (where, for example, there might be 256 distinct characters and no bit reduction is possible).

Run Length Encoding (1)

Original Data

```
aaaaaaaaaaaaabbbbbbbbbbbbbbbccccccccccdddefgggg
```

46 characters, 7 distinct characters

The text above contains only 7 distinct characters, so we could try encoding it with 3 bits per character instead of 8 bits.

Run Length Encoding (2)

Original Data

```
aaaaaaaaaaaaabbbbbbbbbbbbbbbccccccccccdddefgggg
```

46 characters, 7 distinct characters

a	000	b	001	c	010	d	011
e	100	f	101	g	110		

```
 a   a   a   a   a   a   a   a   a   a   a   a   a   b   b   b   b   b
000 000 000 000 000 000 000 000 000 000 000 000 000 001 001 001 001 001
 b   b   b   b   b   b   b   b   b   c   c   c   c   c   c   c   c   c
001 001 001 001 001 001 001 001 001 010 010 010 010 010 010 010 010 010
 c   d   d   d   e   f   g   g   g   g
010 011 011 011 100 101 110 110 110 110
```

138 bits (18 bytes)

The text above contains only 7 distinct characters, so we could try encoding it with 3 bits per character instead of 8 bits.

Run Length Encoding (3)

Original Data

```
aaaaaaaaaaaaabbbbbbbbbbbbbbbccccccccccdddefgggg
```

46 characters, 7 distinct characters

```
  13       a       14       b       10       c       3
00001101 01100001 00001110 01100010 00001010 01100011 00000011
   d       1       e       1       f       4       g
01100100 00000001 01100101 00000001 01100110 00000100 01100111
```

112 bits (14 bytes)

Even with the original 8-bit encoding of the text, we can achieve compression using **run length encoding**. Notice that the input data contains several long runs of a single character. We can condense these runs into a two byte (count, character) pair, as in the example above.

Run Length Encoding (4)

Original Data

```
aaaaaaaaaaaaabbbbbbbbbbbbbbbccccccccccdddefgggg
```

46 characters, 7 distinct characters

a	000	b	001	c	010	d	011
e	100	f	101	g	110		

```
  13     a     14     b     10     c      3     d      1     e
00001101 000 00001110 001 00001010 010 00000011 011 00000001 100
   1     f      4     g
00000001 101 00000100 110
```

77 bits (10 bytes)

We can save a few more bytes by combining the 3-bit encoding with run length encoding. Notice that the run length is still specified in 8 bits (since a 3-bit run length would only allow $2^3 = 8$ characters per run).

Run Length Encoding (5)

Original Data

```
aaaaaaaaaaaaabbbbbbbbbbbbbbbccccccccccdddefgggg
```

46 characters, 7 distinct characters

a	000	b	001	c	010	d	011
e	100	f	101	g	110		

```
  13     a     14     b     10     c      3     d      1     e
00001101 000 00001110 001 00001010 010 00000011 011 00000001 100
   1     f      4     g
00000001 101 00000100 110
```

77 bits (10 bytes)

Question: Will run length encoding always improve compression?

Run Length Encoding (6)

Original Data

```
abcdabcdabcdefg
```

15 characters, 7 distinct characters

a	000	b	001	c	010	d	011
e	100	f	101	g	110		

```
   1      a      1      b      1      c      1      d      1      a
00000001 000 00000001 001 00000001 010 00000001 011 00000001 000
   1      b      1      c      1      d      1      a      1      b
00000001 001 00000001 010 00000001 011 00000001 000 00000001 001
   1      c      1      d      1      e      1      f      1      g
00000001 010 00000001 011 00000001 100 00000001 101 00000001 110
```

165 bits (21 bytes)

Answer: No. When there are runs of very short lengths, the overhead of run length encoding can result in the encoded version requiring more space than the original.

Run Length Encoding (7)

Original Data

```
abcdabcdabcdefg
```

15 characters, 7 distinct characters

a	000	b	001	c	010	d	011
e	100	f	101	g	110		

```
   1      a      1      b      1      c      1      d      1      a
00000001 000 00000001 001 00000001 010 00000001 011 00000001 000
   1      b      1      c      1      d      1      a      1      b
00000001 001 00000001 010 00000001 011 00000001 000 00000001 001
   1      c      1      d      1      e      1      f      1      g
00000001 010 00000001 011 00000001 100 00000001 101 00000001 110
```

165 bits (21 bytes)

In some specific applications (e.g. image compression), run length encoding can be very practical, since the data can be expected to contain long runs.

Run Length Encoding (8)

Original Data

```
abcdabcdabcdefg
```

15 characters, 7 distinct characters

a	000	b	001	c	010	d	011
e	100	f	101	g	110		

```
   1      a      1      b      1      c      1      d      1      a
00000001 000 00000001 001 00000001 010 00000001 011 00000001 000
   1      b      1      c      1      d      1      a      1      b
00000001 001 00000001 010 00000001 011 00000001 000 00000001 001
   1      c      1      d      1      e      1      f      1      g
00000001 010 00000001 011 00000001 100 00000001 101 00000001 110
```

165 bits (21 bytes)

However, for arbitrary data, we cannot assume that the data will contain runs of duplicate characters.

Developing a Scheme (1)

Original Data

```
aaaaaaaaaaaaabbbbbbbbbbbbbbbccccccccccdddefgggg
```

46 characters, 7 distinct characters

a	000	b	001	c	010	d	011
e	100	f	101	g	110		

```
 a   a   a   a   a   a   a   a   a   a   a   a   a   b   b   b   b   b
000 000 000 000 000 000 000 000 000 000 000 000 000 001 001 001 001 001
 b   b   b   b   b   b   b   b   b   c   c   c   c   c   c   c   c   c
001 001 001 001 001 001 001 001 001 010 010 010 010 010 010 010 010 010
 c   d   d   d   e   f   g   g   g   g
010 011 011 011 100 101 110 110 110 110
```

138 bits (18 bytes)

We will focus on ways to encode each character (or, later, group of characters) using a smaller number of bits.

Developing a Scheme (2)

Original Data

```
aaaaaaaaaaaaabbbbbbbbbbbbbbbccccccccccdddefgggg
```

46 characters, 7 distinct characters

a	000	b	001	c	010	d	011
e	100	f	101	g	110		

```
 a   a   a   a   a   a   a   a   a   a   a   a   a   b   b   b   b   b
000 000 000 000 000 000 000 000 000 000 000 000 000 001 001 001 001 001
 b   b   b   b   b   b   b   b   b   c   c   c   c   c   c   c   c   c
001 001 001 001 001 001 001 001 001 010 010 010 010 010 010 010 010 010
 c   d   d   d   e   f   g   g   g   g
010 011 011 011 100 101 110 110 110 110
```

138 bits (18 bytes)

Observation: Each character is encoded in 3 bits. However, some characters only appear once or twice, while others appear very frequently.

Developing a Scheme (3)

Original Data

```
aaaaaaaaaaaaabbbbbbbbbbbbbbbccccccccccdddefgggg
```

46 characters, 7 distinct characters

a	11	b	10	c	01	d	001
e	100	f	101	g	00		

```
a  a  a  a  a  a  a  a  a  a  a  a  a  b  b  b  b  b  b  b  b  b  b  b  b
11 11 11 11 11 11 11 11 11 11 11 11 11 10 10 10 10 10 10 10 10 10 10 10 10
b  b  c  c  c  c  c  c  c  c  c  c  d  d  d  e  f  g  g  g  g
10 10 01 01 01 01 01 01 01 01 01 01 001 001 001 100 101 00 00 00 00
```

97 bits (13 bytes)

Idea: Construct an encoding where some characters are encoded into 3 bits and some are encoded into only 2 bits.

Developing a Scheme (4)

Original Data

```
bbcgbbcccbaacabcadaacebbabaaccbbgddgcbfcababag
```

46 characters, 7 distinct characters

a	11	b	10	c	01	d	001
e	100	f	101	g	00		

```
 b  b  c  g  b  b  c  c  c  b  a  a  c  a  b  c  a  d  a  a  c  e  b  b
10 10 01 00 10 10 01 01 01 10 11 11 01 11 10 01 11 001 11 11 01 100 10 10
 a  b  a  a  c  c  b  b  g  d  d  g  c  b  f  c  a  b  a  b  a  g
11 10 11 11 01 01 10 10 00 001 001 00 01 10 101 01 11 10 11 10 11 00
```

97 bits (13 bytes)

Since the order of characters is irrelevant, the example above (which has the same characters as the previous example, but in a different order) has the same compressed size.

Developing a Scheme (5)

Original Data

```
bbcgbbcccbaacabcadaacebbabaaccbbgddgcbfcababag
```

46 characters, 7 distinct characters

a	11	b	10	c	01	d	001
e	100	f	101	g	00		

```
 b  b  c  g  b  b  c  c  c  b  a  a  c  a  b  c  a  d  a  a  c  e  b  b
10 10 01 00 10 10 01 01 01 10 11 11 01 11 10 01 11 001 11 11 01 100 10 10
 a  b  a  a  c  c  b  b  g  d  d  g  c  b  f  c  a  b  a  b  a  g
11 10 11 11 01 01 10 10 00 001 001 00 01 10 101 01 11 10 11 10 11 00
```

97 bits (13 bytes)

Question: The first six bits of the compressed data are `101001`. To decompress, do we decode this to `bbc` (`10 10 01`) or to `fd` (`101 001`)?

Developing a Scheme (6)

Original Data

```
bbcgbbcccbaacabcadaacebbabaaccbbgddgcbfcababag
```

46 characters, 7 distinct characters

a	11	b	10	c	01	d	001
e	100	f	101	g	00		

```
 b  b  c  g  b  b  c  c  c  b  a  a  c  a  b  c  a  d  a  a  c  e  b  b
10 10 01 00 10 10 01 01 01 10 11 11 01 11 10 01 11 001 11 11 01 100 10 10
 a  b  a  a  c  c  b  b  g  d  d  g  c  b  f  c  a  b  a  b  a  g
11 10 11 11 01 01 10 10 00 001 001 00 01 10 101 01 11 10 11 10 11 00
```

97 bits (13 bytes)

Answer: It is impossible to differentiate (the spaces shown in the compressed data listing above are obviously not present in the output file). This is a serious problem (since we must be assured that the original data can be restored).

Developing a Scheme (7)

Original Data

```
bbcgbbcccbaacabcadaacebbabaaccbbgddgcbfcababag
```

46 characters, 7 distinct characters

a	11	b	10	c	01	d	001
e	100	f	101	g	00		

```
 b  b  c  g  b  b  c  c  c  b  a  a  c  a  b  c  a  d  a  a  c  e  b  b
10 10 01 00 10 10 01 01 01 10 11 11 01 11 10 01 11 001 11 11 01 100 10 10
 a  b  a  a  c  c  b  b  g  d  d  g  c  b  f  c  a  b  a  b  a  g
11 10 11 11 01 01 10 10 00 001 001 00 01 10 101 01 11 10 11 10 11 00
```

97 bits (13 bytes)

The ultimate problem is that there are some encodings whose bit sequence is a **prefix** of another encoding.

Developing a Scheme (8)

Original Data

```
bbcgbbcccbaacabcadaacebbabaaccbbgddgcbfcababag
```

46 characters, 7 distinct characters

a	11	b	10	c	01	d	001
e	100	f	101	g	00		

```
 b  b  c  g  b  b  c  c  c  b  a  a  c  a  b  c  a  d  a  a  c  e  b  b
10 10 01 00 10 10 01 01 01 10 11 11 01 11 10 01 11 001 11 11 01 100 10 10
 a  b  a  a  c  c  b  b  g  d  d  g  c  b  f  c  a  b  a  b  a  g
11 10 11 11 01 01 10 10 00 001 001 00 01 10 101 01 11 10 11 10 11 00
```

97 bits (13 bytes)

Suppose you see the bit sequence 10. Should you stop and decode it as 'b' or continue reading and possibly decode it as 'e' (100) or 'f' (101)?

Developing a Scheme (9)

Original Data

```
bbcgbbcccbaacabcadaacebbabaaccbbgddgcbfcababag
```

46 characters, 7 distinct characters

a	11	b	10	c	01	d	001
e	100	f	101	g	00		

```
 b  b  c  g  b  b  c  c  c  b  a  a  c  a  b  c  a  d  a  a  c  e  b  b
10 10 01 00 10 10 01 01 01 10 11 11 01 11 10 01 11 001 11 11 01 100 10 10
 a  b  a  a  c  c  b  b  g  d  d  g  c  b  f  c  a  b  a  b  a  g
11 10 11 11 01 01 10 10 00 001 001 00 01 10 101 01 11 10 11 10 11 00
```

97 bits (13 bytes)

Since there is no way to know, this encoding cannot achieve lossless compression, since it would be impossible to decode the result.

Prefix-Free Codes (1)

Original Data

```
bbcgbbcccbaacabcadaacebbabaaccbbgddgcbfcababag
```

46 characters, 7 distinct characters

a	00	b	01	c	10	d	1110
e	11110	f	11111	g	110		

```
 b  b  c  g  b  b  c  c  c  b  a  a  c  a  b  c  a  d  a  a  c  e
01 01 10 110 01 01 10 10 10 01 00 00 10 00 01 10 00 1110 00 00 10 11110
 b  b  a  b  a  a  c  c  b  b  g  d  d  g  c  b  f  c  a  b  a
01 01 00 01 00 00 10 10 01 01 110 1110 1110 110 10 01 11111 10 00 01 00
 b  a  g
01 00 110
```

108 bits (14 bytes)

However, it is possible to find an encoding which is **prefix free**. Notice that no bit sequence in the table is a prefix of another bit sequence.

Prefix-Free Codes (2)

Original Data

```
bbcgbbcccbaacabcadaacebbabaaccbbgddgcbfcababag
```

46 characters, 7 distinct characters

a	00	b	01	c	10	d	1110
e	11110	f	11111	g	110		

```
 b  b  c  g  b  b  c  c  c  b  a  a  c  a  b  c  a  d  a  a  c  e
01 01 10 110 01 01 10 10 10 01 00 00 10 00 01 10 00 1110 00 00 10 11110
 b  b  a  b  a  a  c  c  b  b  g  d  d  g  c  b  f  c  a  b  a
01 01 00 01 00 00 10 10 01 01 110 1110 1110 110 10 01 11111 10 00 01 00
 b  a  g
01 00 110
```

108 bits (14 bytes)

An encoding scheme which is prefix free is called a **prefix-free code** or, somewhat counterintuitively, a **prefix code**.

Prefix-Free Codes (3)

Original Data

```
bbcgbbcccbaacabcadaacebbabaaccbbgddgcbfcababag
```

46 characters, 7 distinct characters

a	00	b	01	c	10	d	1110
e	11110	f	11111	g	110		

```
 b  b  c  g  b  b  c  c  c  b  a  a  c  a  b  c  a  d  a  a  c  e
01 01 10 110 01 01 10 10 10 01 00 00 10 00 01 10 00 1110 00 00 10 11110
 b  b  a  b  a  a  c  c  b  b  g  d  d  g  c  b  f  c  a  b  a
01 01 00 01 00 00 10 10 01 01 110 1110 1110 110 10 01 11111 10 00 01 00
 b  a  g
01 00 110
```

108 bits (14 bytes)

Therefore, if we read enough bits to match any character, we know to decode that character.

Prefix-Free Codes (4)

Original Data

```
bbcgbbcccbaacabcadaacebbabaaccbbgddgcbfcababag
```

46 characters, 7 distinct characters

a	00	b	01	c	10	d	1110
e	11110	f	11111	g	110		

```
 b  b  c  g  b  b  c  c  c  b  a  a  c  a  b  c  a  d  a  a  c  e
01 01 10 110 01 01 10 10 10 01 00 00 10 00 01 10 00 1110 00 00 10 11110
 b  b  a  b  a  a  c  c  b  b  g  d  d  g  c  b  f  c  a  b  a
01 01 00 01 00 00 10 10 01 01 110 1110 1110 110 10 01 11111 10 00 01 00
 b  a  g
01 00 110
```

108 bits (14 bytes)

For example, if we read the bits 10, we must decode it to ‘c’ (and then start on the next character). If we read the bits 11, we must read more bits (to produce either ‘d’, ‘e’, ‘f’ or ‘g’).

Prefix-Free Codes (5)

Original Data

```
bbcgbbcccbaacabcadaacebbabaaccbbgddgcbfcababag
```

46 characters, 7 distinct characters

a	00	b	01	c	10	d	1110
e	11110	f	11111	g	110		

```
 b  b  c  g  b  b  c  c  c  b  a  a  c  a  b  c  a  d  a  a  c  e
01 01 10 110 01 01 10 10 10 01 00 00 10 00 01 10 00 1110 00 00 10 11110
 b  b  a  b  a  a  c  c  b  b  g  d  d  g  c  b  f  c  a  b  a
01 01 00 01 00 00 10 10 01 01 110 1110 1110 110 10 01 11111 10 00 01 00
 b  a  g
01 00 110
```

108 bits (14 bytes)

Notice that this encoding compresses the data to 14 bytes. The last valid encoding we saw (the fixed 3-bit encoding) compressed the data to 18 bytes.

Prefix-Free Codes (6)

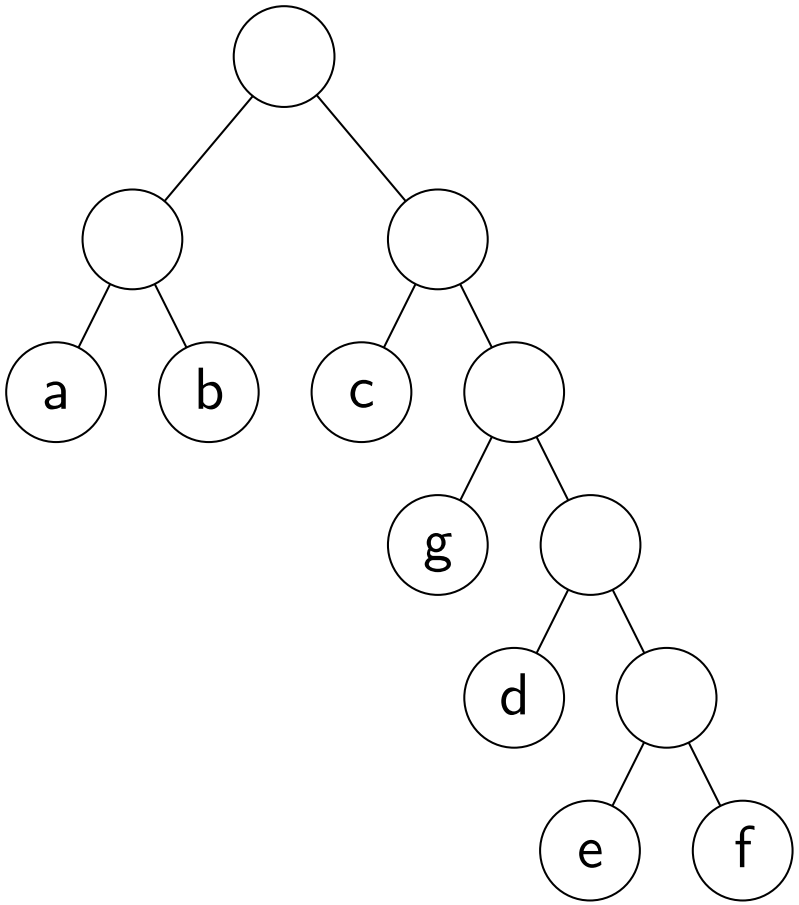
Important Qualifier: If a compression scheme uses a specialized encoding of characters to bit sequences, usually the conversion table has to be stored along with the compressed data (so that the decoder will recognize the encoding).

Storing the conversion table (or some equivalent representation) will inflate the compressed size (and can sometimes negate the compression advantage of more complicated schemes). However, if the size of the conversion table can be kept small (or constant) relative to the size of the input data, the overhead will be less significant as the input data size grows.

Encodings as Trees (1)

Question: How can we easily tell whether a particular encoding (of letters or sequences to binary strings) is prefix-free?

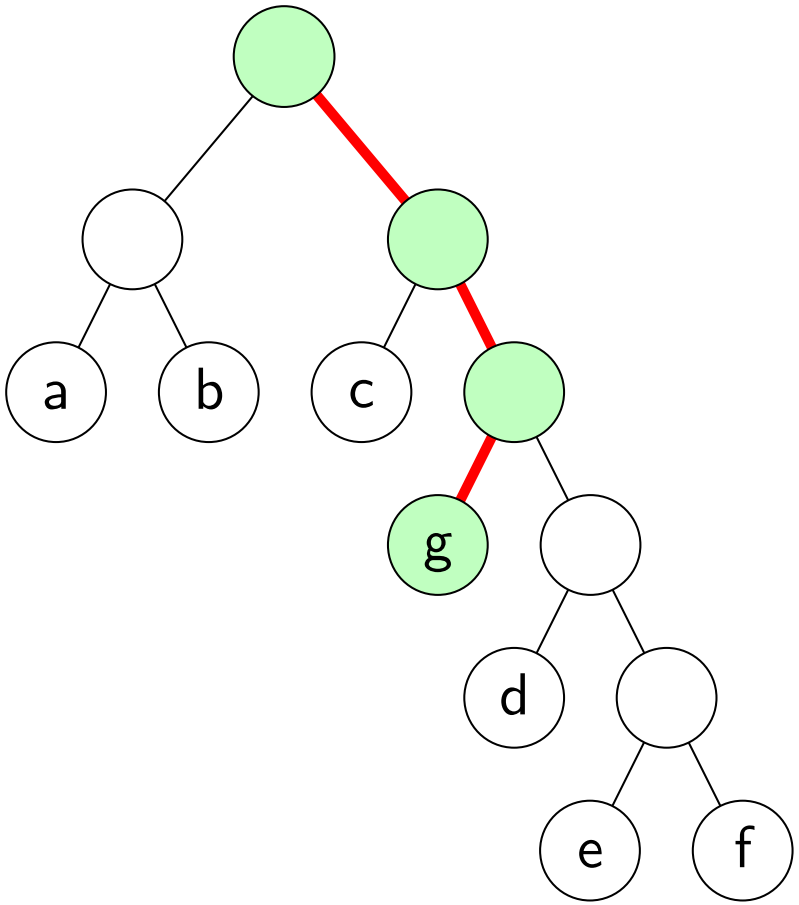
Encodings as Trees (2)



a	00
b	01
c	10
d	1110
e	11110
f	11111
g	110

An encoding scheme (into binary) can be represented by a binary tree.

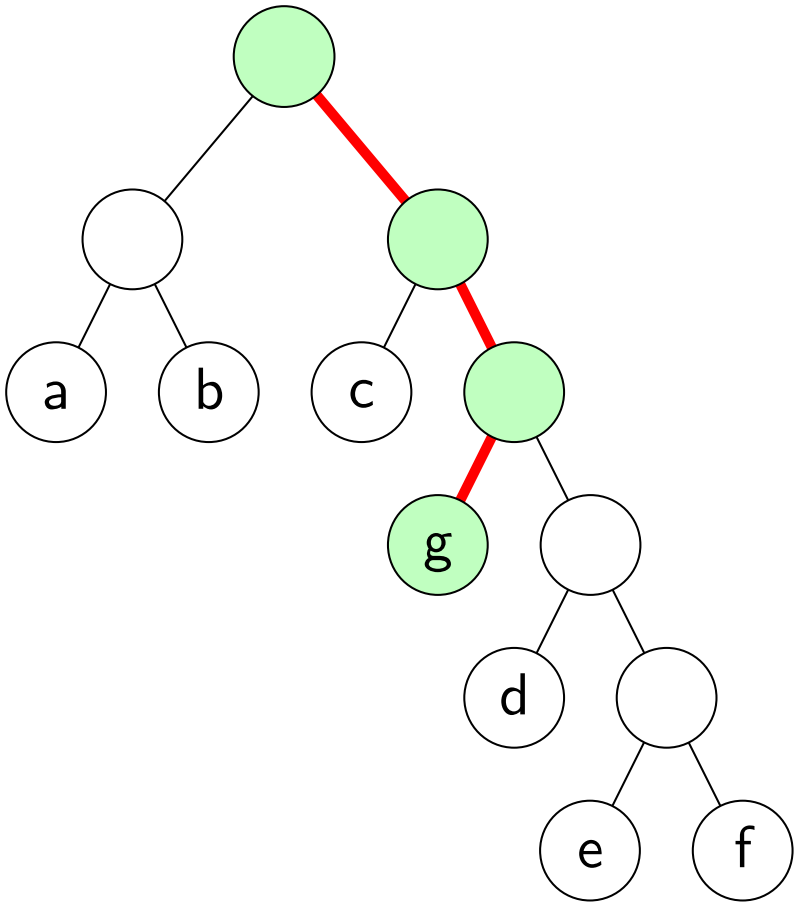
Encodings as Trees (3)



a	00
b	01
c	10
d	1110
e	11110
f	11111
g	110

Notice that the path from the root to the node labelled ‘g’ is ‘Right, Right, Left’. We can encode such a path into binary by using 0 to represent a left turn and 1 to represent a right turn.

Encodings as Trees (4)



a	00
b	01
c	10
d	1110
e	11110
f	11111
g	110

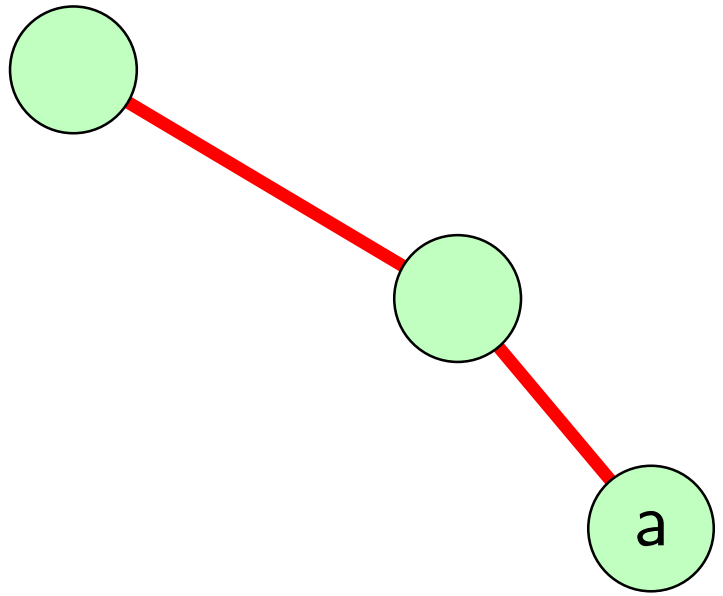
This gives a natural correspondence between trees and binary sequences.

Encodings as Trees (5)

a	11
b	10
c	01
d	001
e	100
f	101
g	00

Task: Construct the binary tree representation of the encoding above. (This is the invalid code from earlier).

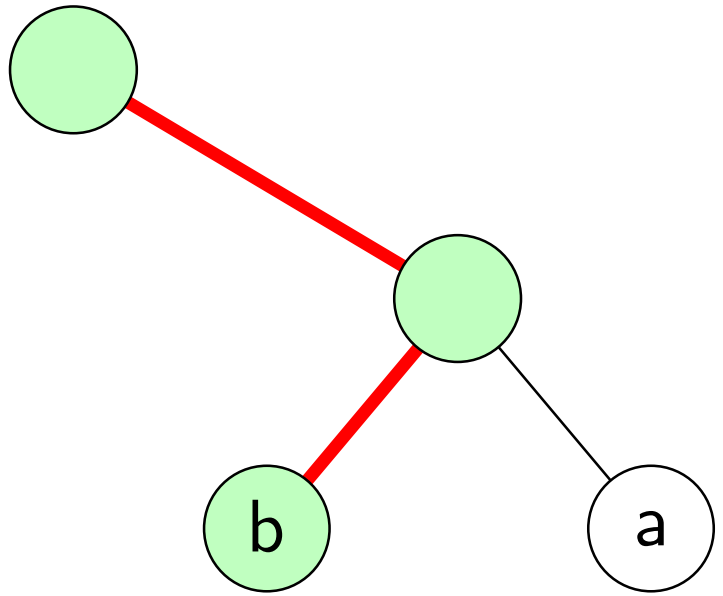
Encodings as Trees (6)



a	11
b	10
c	01
d	001
e	100
f	101
g	00

We can build the tree symbol-by-symbol, creating any nodes necessary to make the tree structure match the binary sequence.

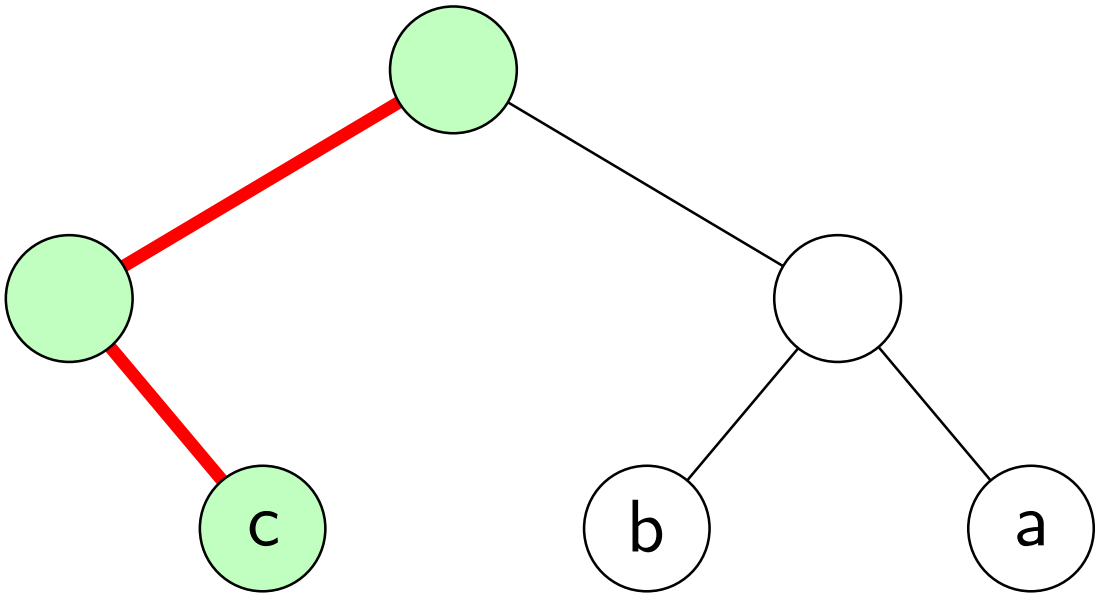
Encodings as Trees (7)



a	11
b	10
c	01
d	001
e	100
f	101
g	00

We can build the tree symbol-by-symbol, creating any nodes necessary to make the tree structure match the binary sequence.

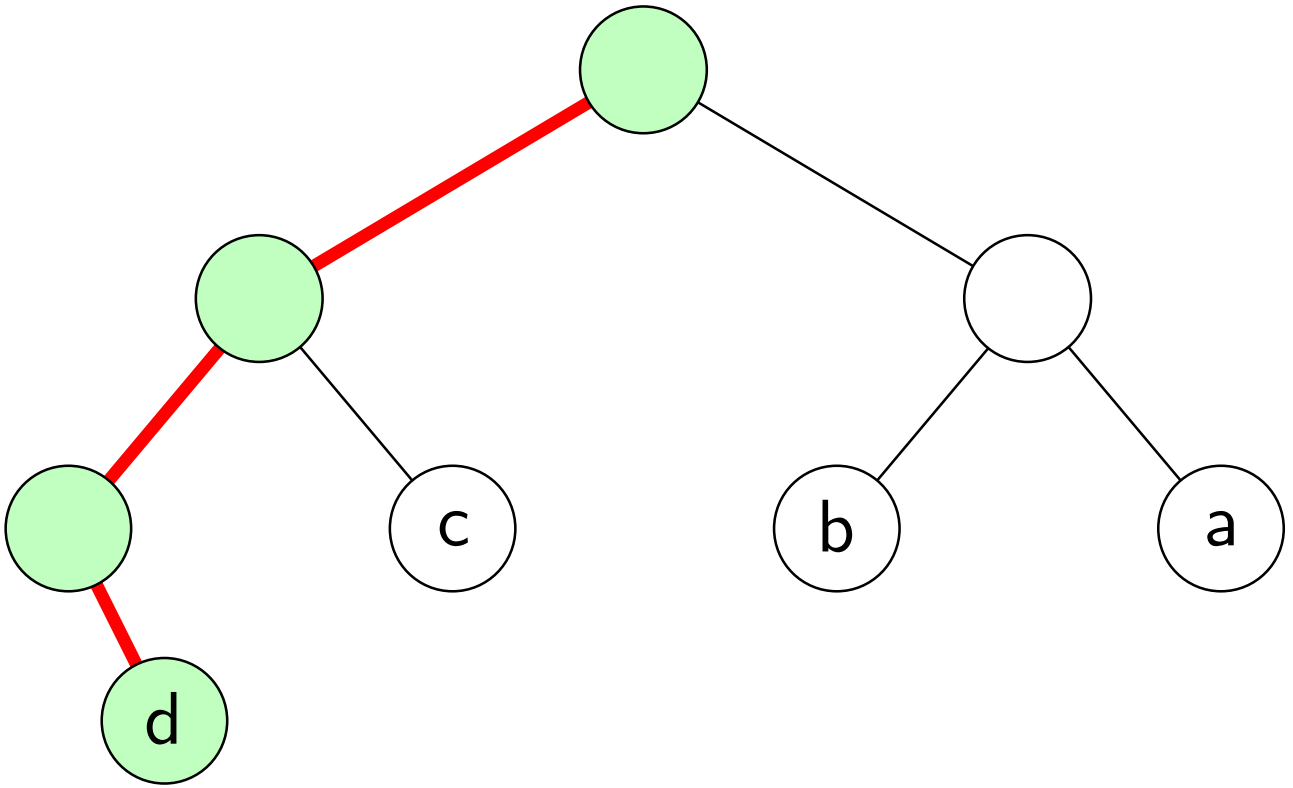
Encodings as Trees (8)



a	11
b	10
c	01
d	001
e	100
f	101
g	00

We can build the tree symbol-by-symbol, creating any nodes necessary to make the tree structure match the binary sequence.

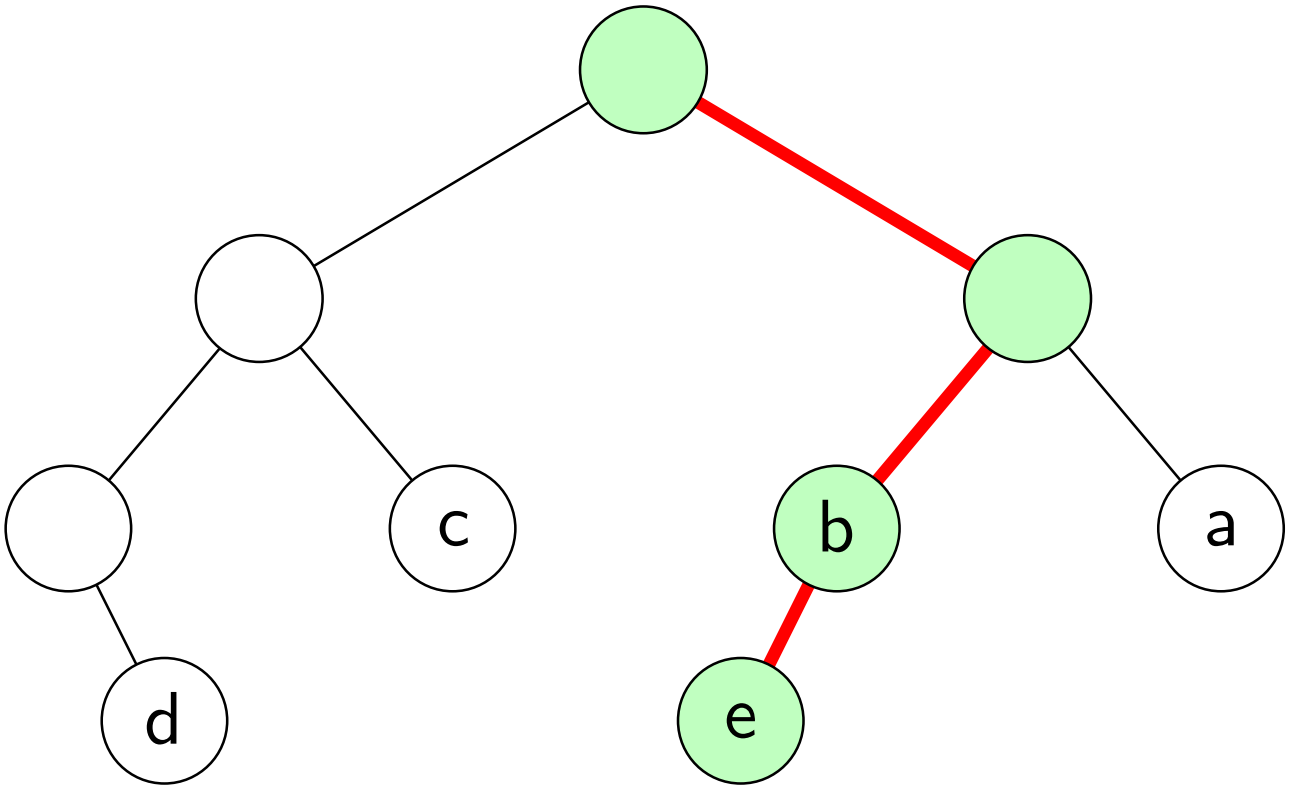
Encodings as Trees (9)



a	11
b	10
c	01
d	001
e	100
f	101
g	00

We can build the tree symbol-by-symbol, creating any nodes necessary to make the tree structure match the binary sequence.

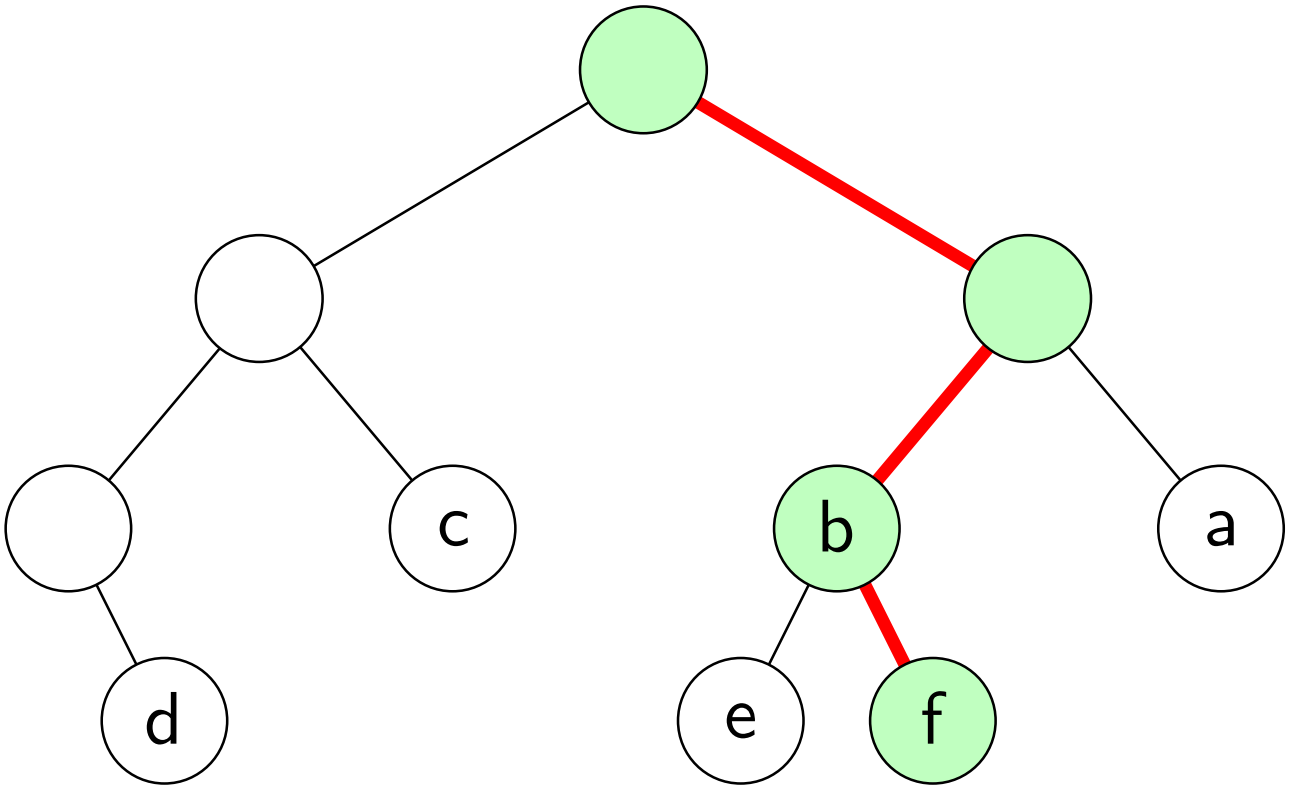
Encodings as Trees (10)



a	11
b	10
c	01
d	001
e	100
f	101
g	00

We can build the tree symbol-by-symbol, creating any nodes necessary to make the tree structure match the binary sequence.

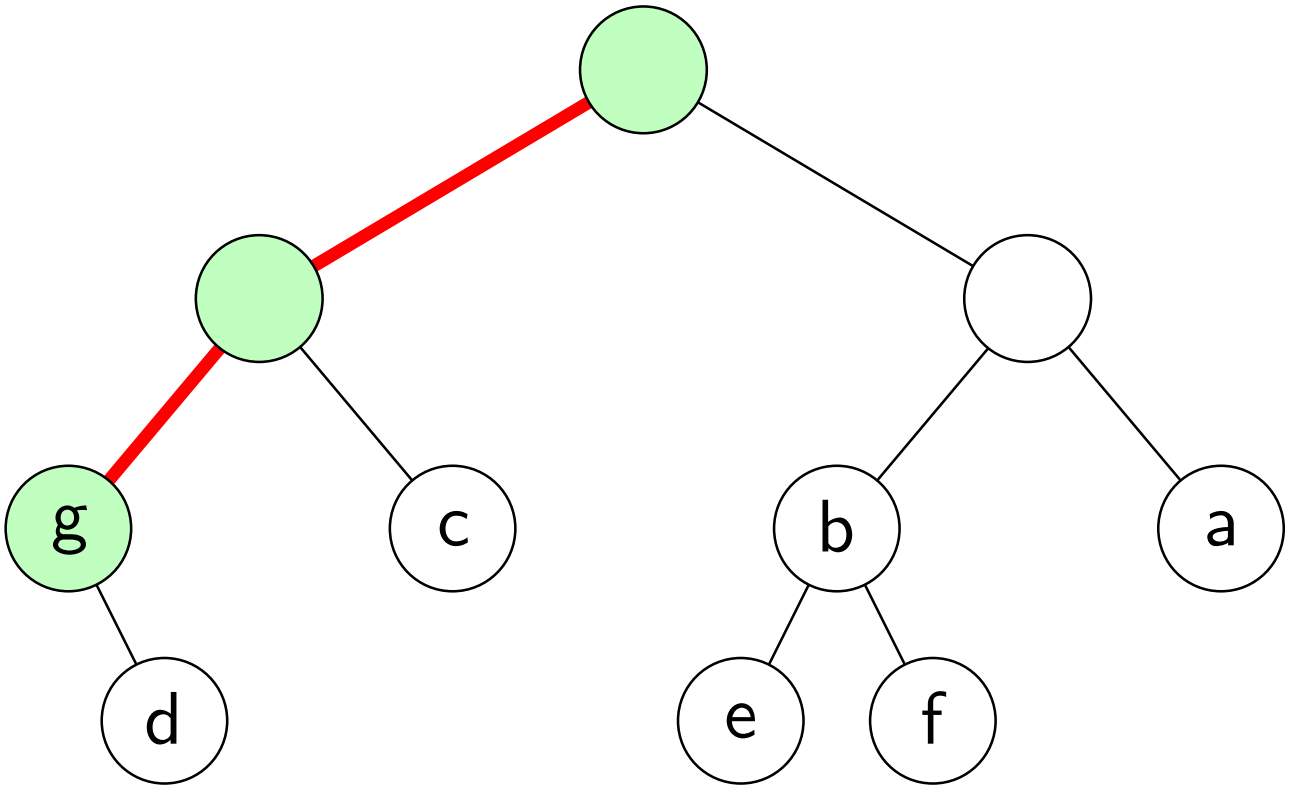
Encodings as Trees (11)



a	11
b	10
c	01
d	001
e	100
f	101
g	00

We can build the tree symbol-by-symbol, creating any nodes necessary to make the tree structure match the binary sequence.

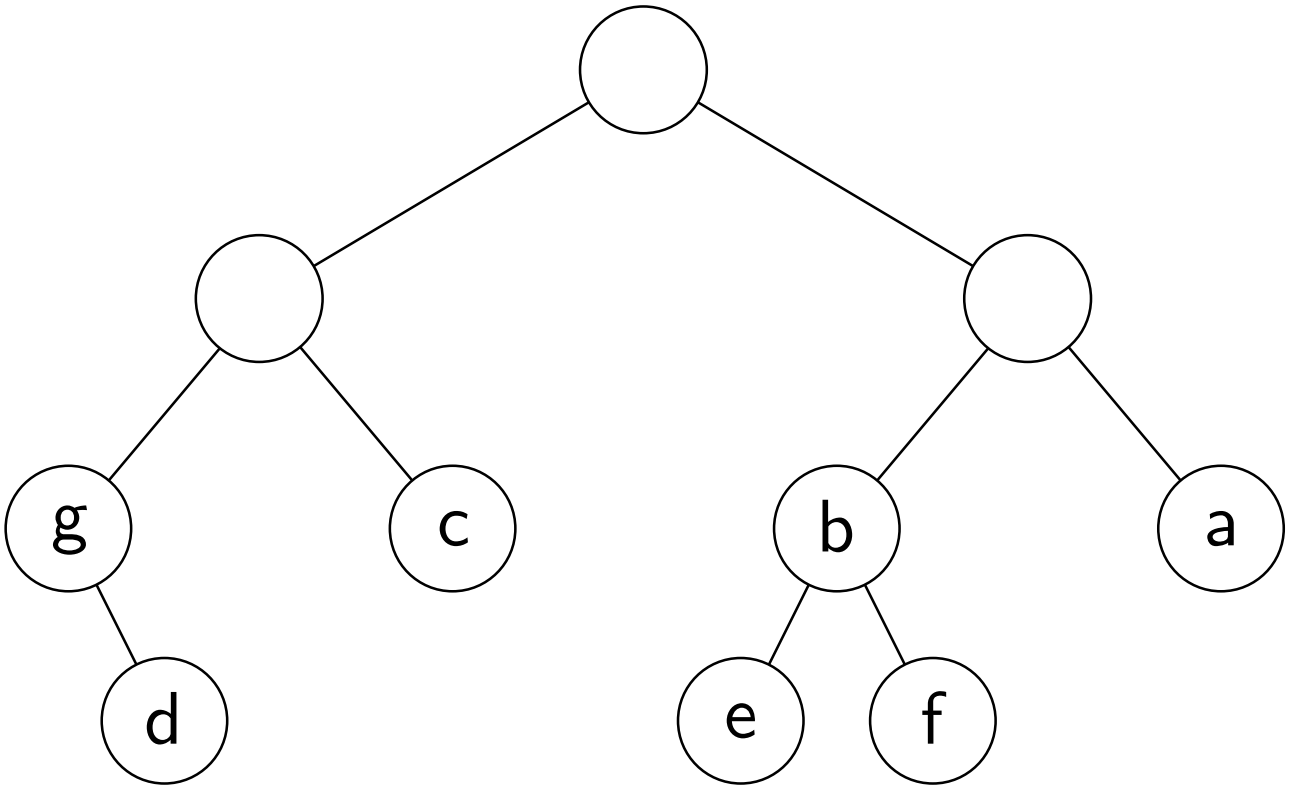
Encodings as Trees (12)



a	11
b	10
c	01
d	001
e	100
f	101
g	00

We can build the tree symbol-by-symbol, creating any nodes necessary to make the tree structure match the binary sequence.

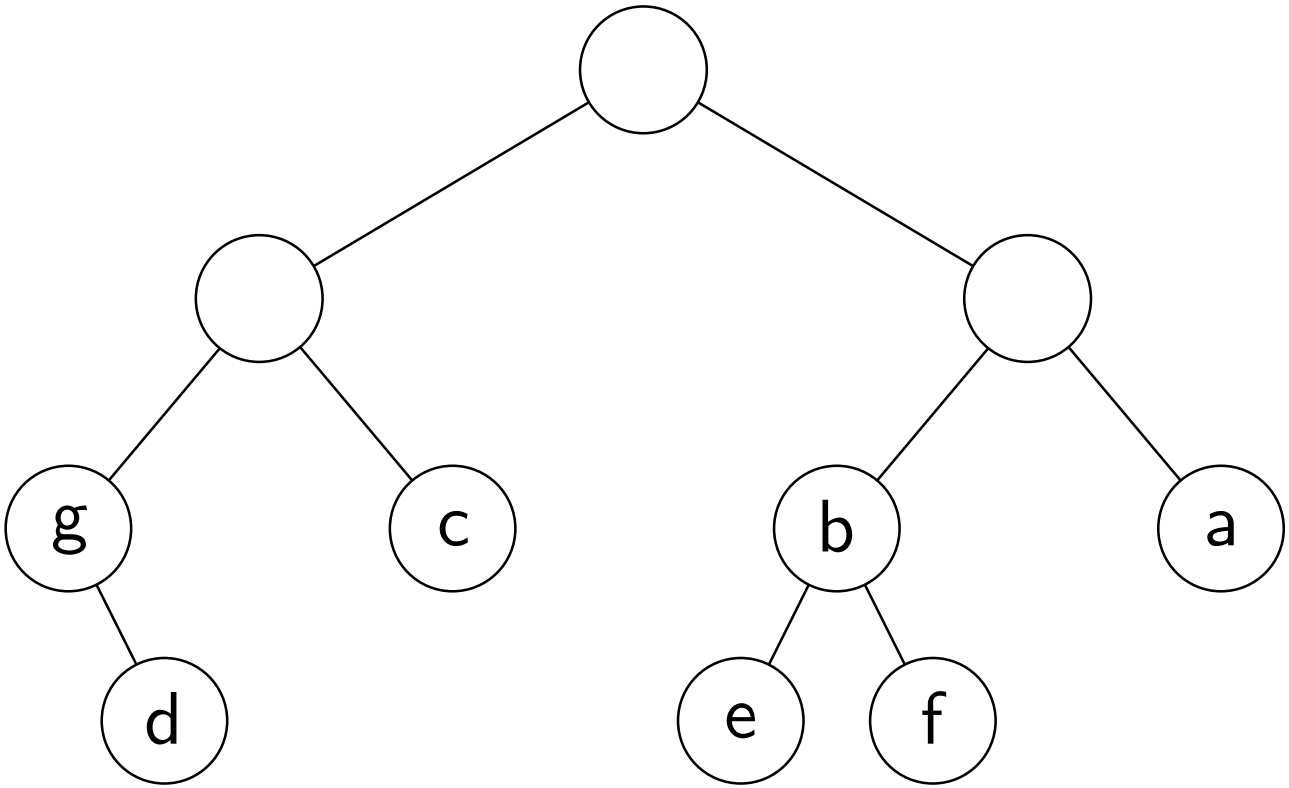
Encodings as Trees (13)



a	11
b	10
c	01
d	001
e	100
f	101
g	00

We know from earlier that this encoding is not prefix-free (and therefore is not suitable for compression). Notice that the encoding for 'b' is a prefix of the encodings for 'e' and 'f'.

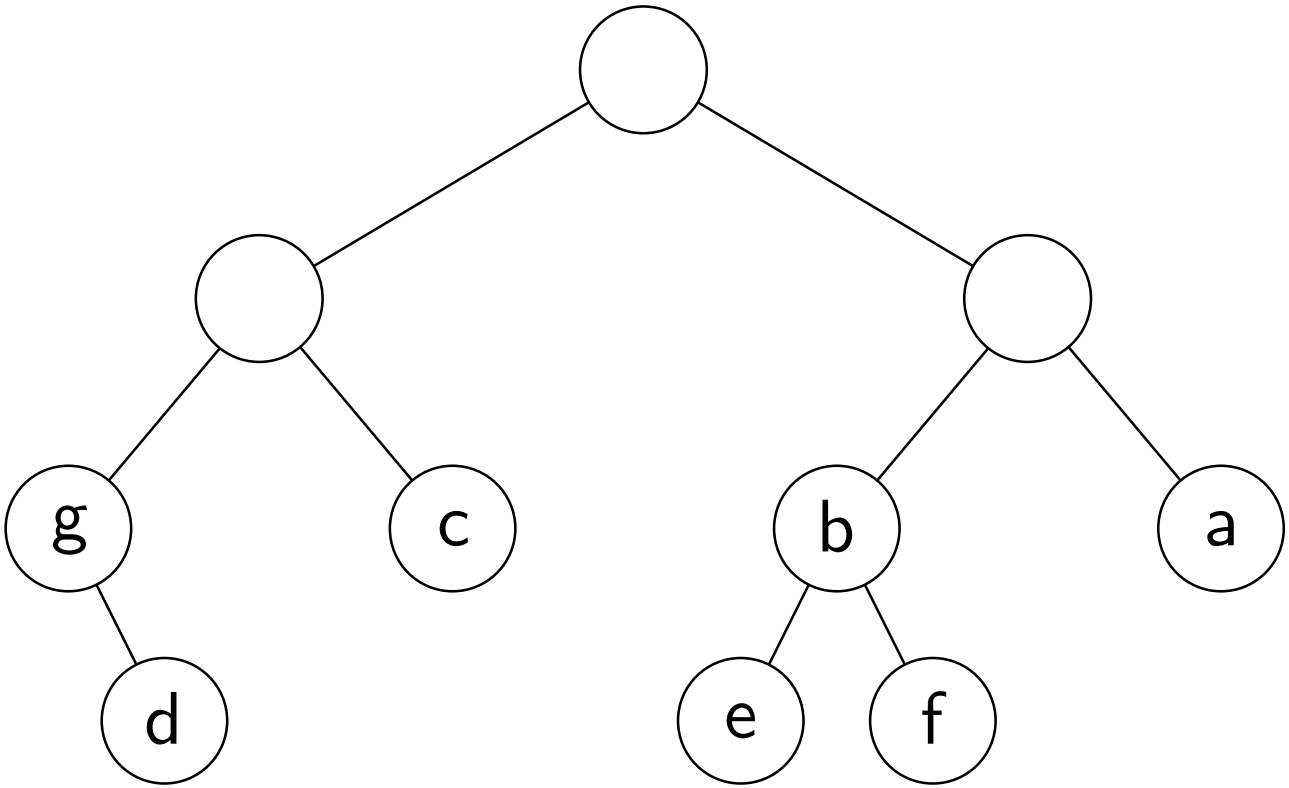
Encodings as Trees (14)



a	11
b	10
c	01
d	001
e	100
f	101
g	00

Fact: An encoding is prefix-free if and only if each encoded binary sequence terminates at a leaf node of the tree.

Encodings as Trees (15)



a	11
b	10
c	01
d	001
e	100
f	101
g	00

(In other words, a prefix-free encoding may not have labels on any internal nodes)

Constructing Encodings (1)

Question: Given a set of symbols (single letters or sequences of text), how can we construct a prefix-free encoding of those symbols into binary?

Constructing Encodings (2)

ab	
bb	
cd	
ae	
pq	
xy	
z	

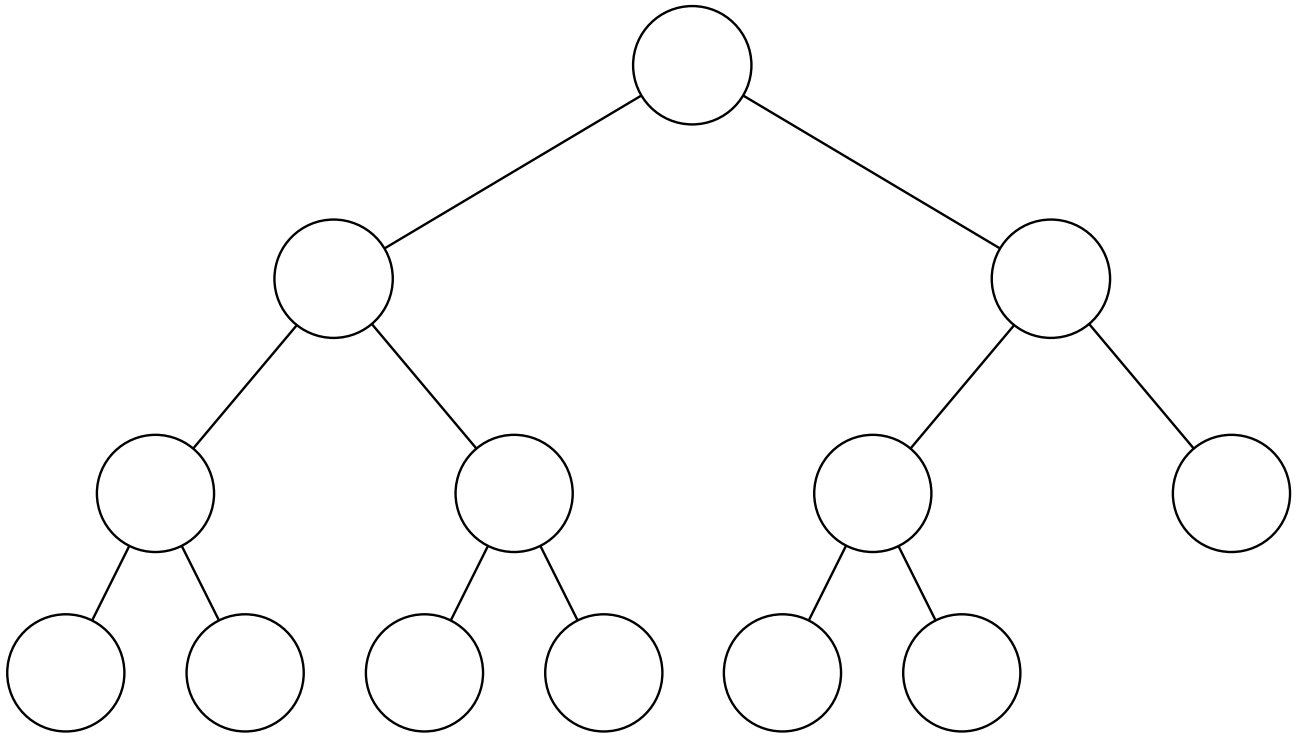
Task: Design a prefix-free encoding for the symbols above.

Constructing Encodings (3)

ab	
bb	
cd	
ae	
pq	
xy	
z	

For a code to be prefix-free, its tree representation must have each encoded symbol in a leaf node.

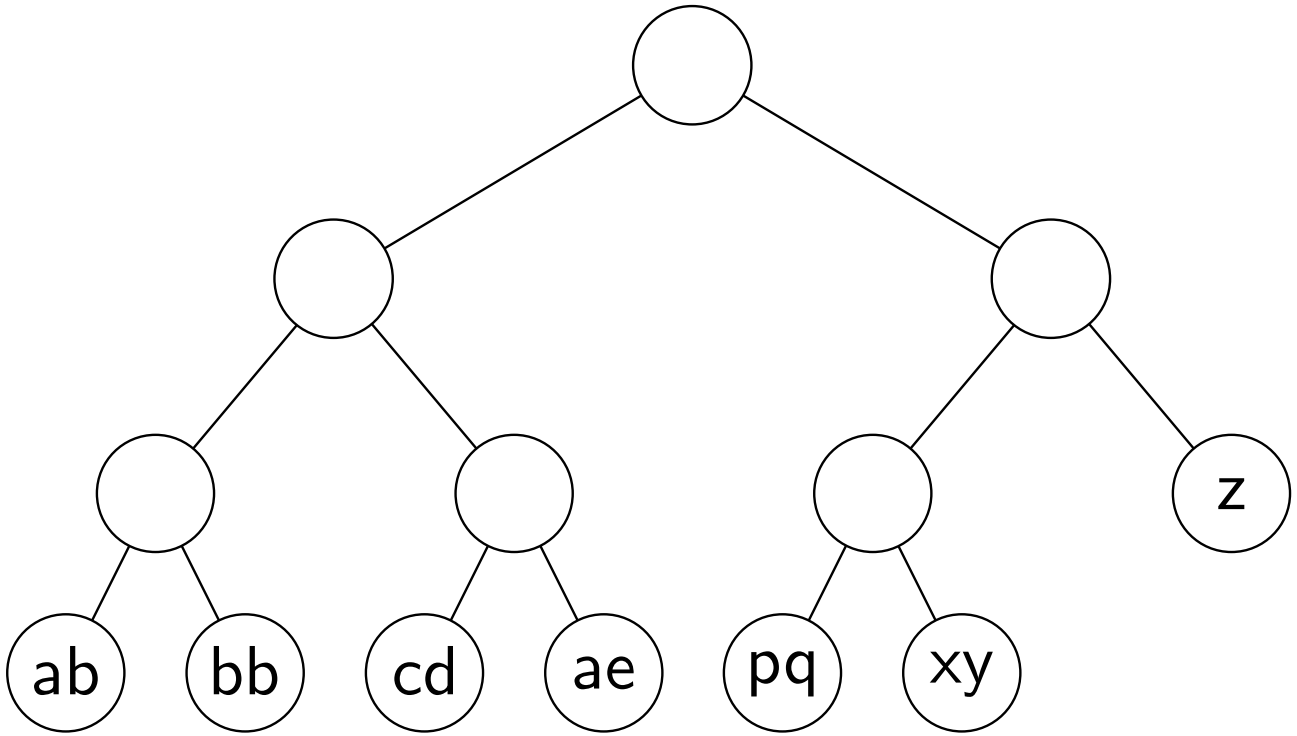
Constructing Encodings (4)



ab	
bb	
cd	
ae	
pq	
xy	
z	

To accomplish this, we could simply create an arbitrary tree with 7 leaf nodes, then assign each symbol to a leaf node. (Note that a 'symbol' may contain multiple characters)

Constructing Encodings (5)



ab	000
bb	001
cd	010
ae	011
pq	100
xy	101
z	11

For example, we could produce the prefix-free encoding above. Obviously, when the shape of the tree and the assignment of symbols to leaves is arbitrary, we cannot guarantee that the encoding will produce good compression.

Huffman Coding (1)

Question: Given a sequence of input data (like a text file) and a set of symbols (like the characters in the file), how can we produce a prefix-free binary encoding that results in high compression?

Huffman Coding (2)

Specifically, we will consider the problem of splitting the text into k character symbols and finding a prefix-free encoding that uses as few bits as possible to encode the entire input file. We therefore desire the following properties.

- Symbols which appear often should have a short encoding (and therefore be located closer to the root of the tree).
- Symbols which appear less often can have longer encodings (and appear further from the root).

Huffman Coding (3)

Original Data

```
raspberry blueberry pear
```

24 characters, 10 distinct characters

One character symbols

Symbol:	_	a	b	e	l	p	r	s	u	y
Frequency:	2	2	3	4	1	2	6	1	1	2

Consider the text fragment above and the histogram showing the number of occurrences of each single character.

Huffman Coding (4)

Original Data

```
raspberry blueberry pear
```

24 characters, 10 distinct characters

One character symbols

Symbol:	_	a	b	e	l	p	r	s	u	y
Frequency:	2	2	3	4	1	2	6	1	1	2

If we design a prefix-free code using the single-character symbols, we would want high-frequency symbols (like 'r' and 'e') to have shorter encodings (and be closer to the root of the tree).

Huffman Coding (5)

Original Data

```
raspberry blueberry pear
```

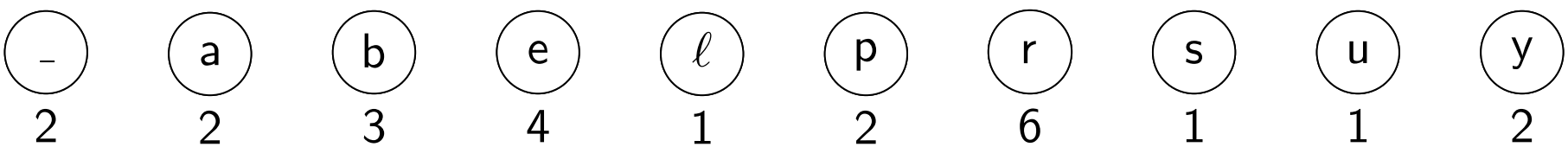
24 characters, 10 distinct characters

One character symbols

Symbol:	_	a	b	e	l	p	r	s	u	y
Frequency:	2	2	3	4	1	2	6	1	1	2

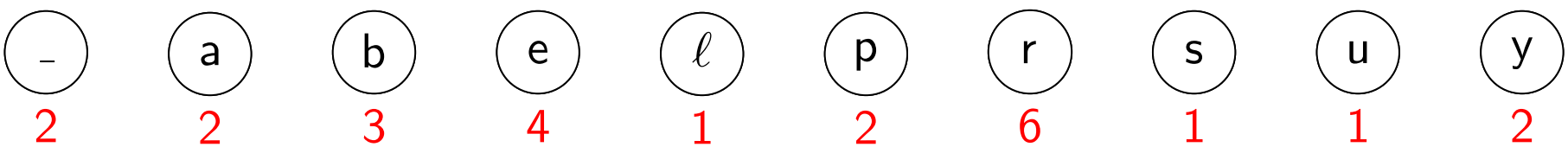
We can use an algorithm called **Huffman Coding** to produce such an encoding. The resulting encoding tree is often called a **Huffman Tree**.

Huffman Coding (6)



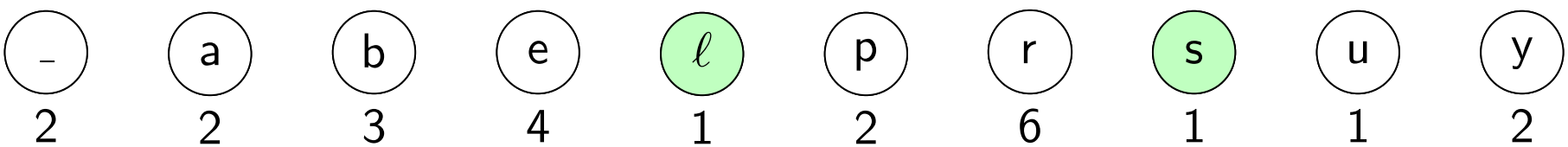
The algorithm builds a single tree by repeatedly merging smaller trees. The algorithm starts with one tree (containing a single node) for each

Huffman Coding (7)



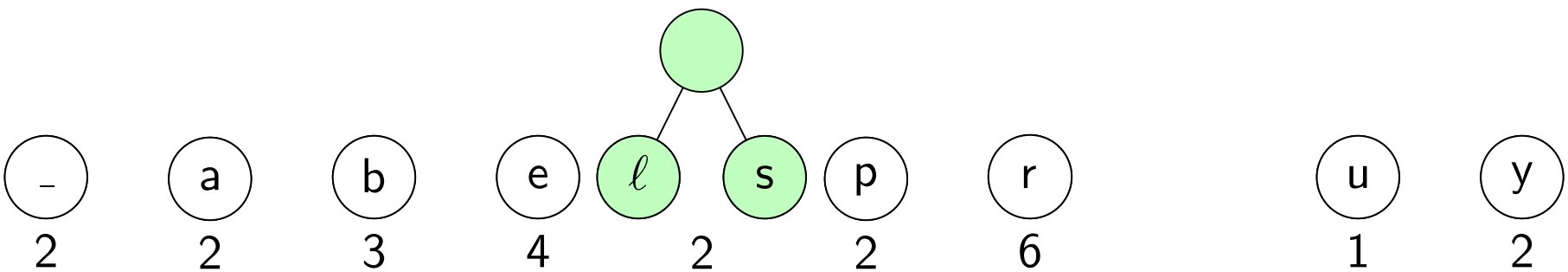
Throughout the algorithm, the **frequency** of each tree is tracked. In cases where we don't know the exact frequency of a symbol, we might instead track a **probability** value. In either case, higher numbers indicate that the symbol appears more often.

Huffman Coding (8)



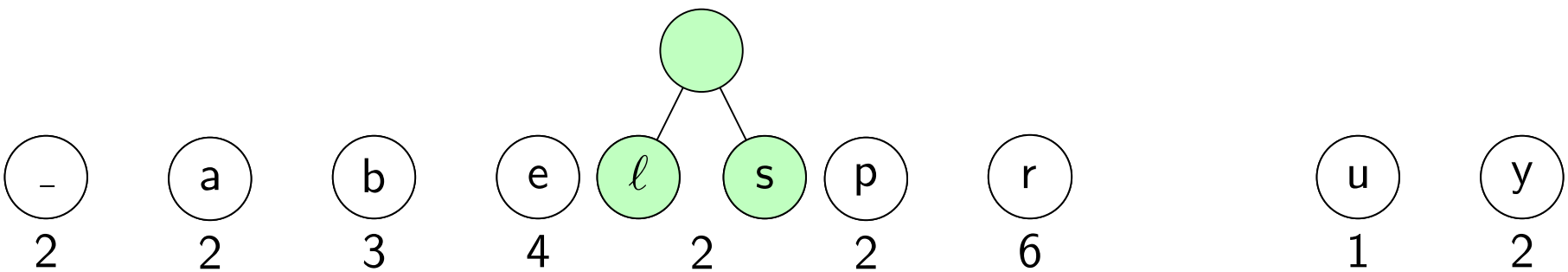
The algorithm will merge trees until only one tree remains. At each step, two trees with the lowest frequency value are chosen...

Huffman Coding (9)



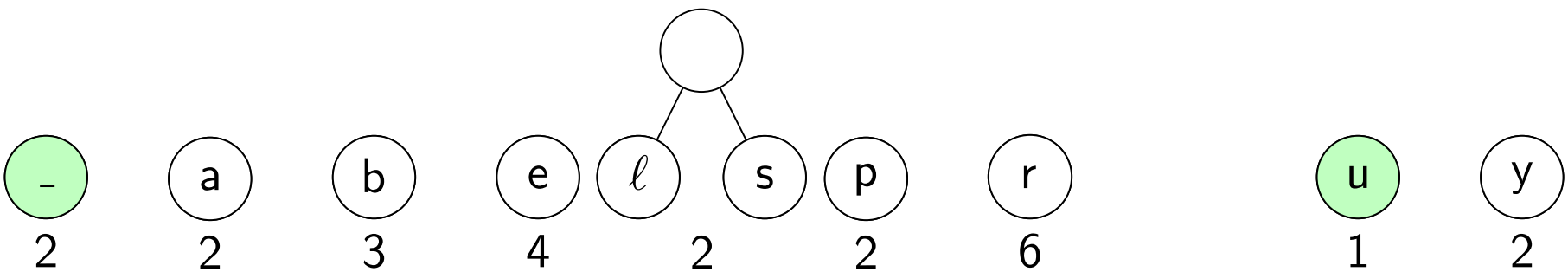
...and merged into a single tree (with frequency equal to the sum of the two merged components).

Huffman Coding (10)



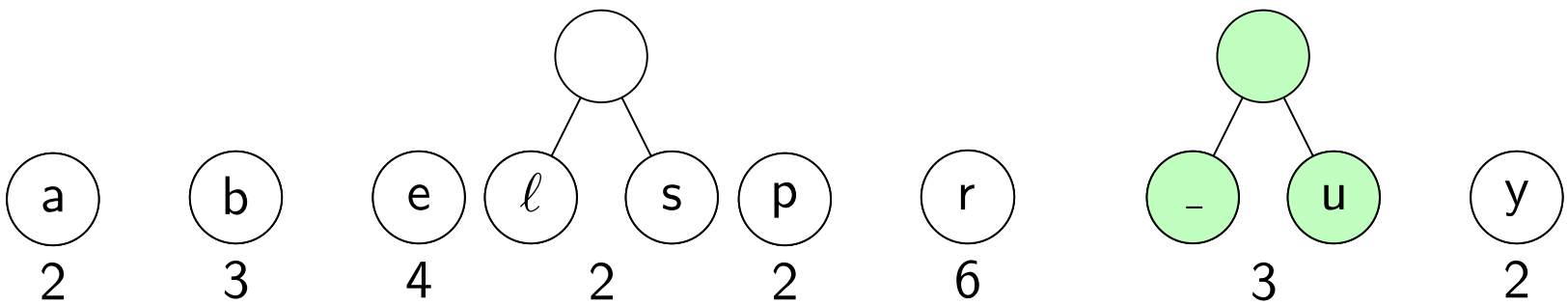
Merging is always done by creating a new root to become the parent of both component trees. Note that the two trees to merge must have the lowest frequencies, but otherwise may be chosen arbitrarily.

Huffman Coding (11)



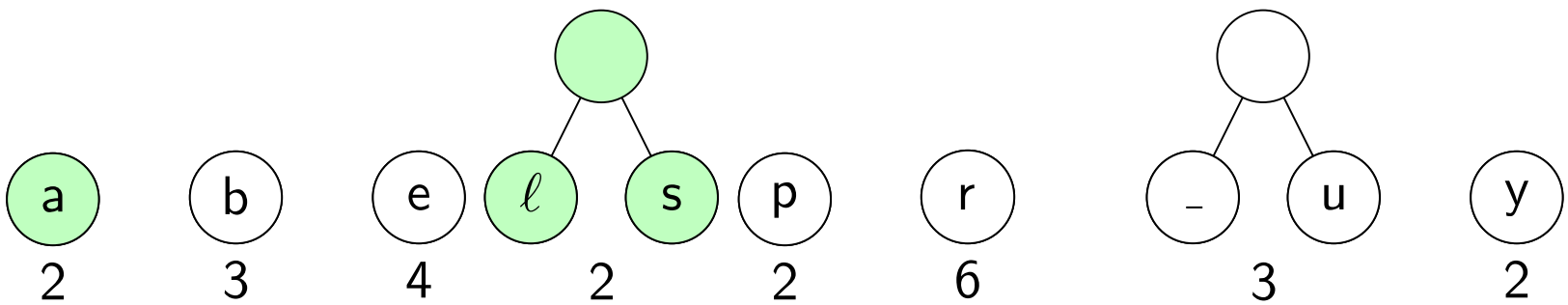
At the next step, two trees with minimum frequency are again chosen. This time, we choose a tree with frequency 1 and a tree with frequency 2. The exact choice of '_' from the various trees with frequency 2 is arbitrary.

Huffman Coding (12)



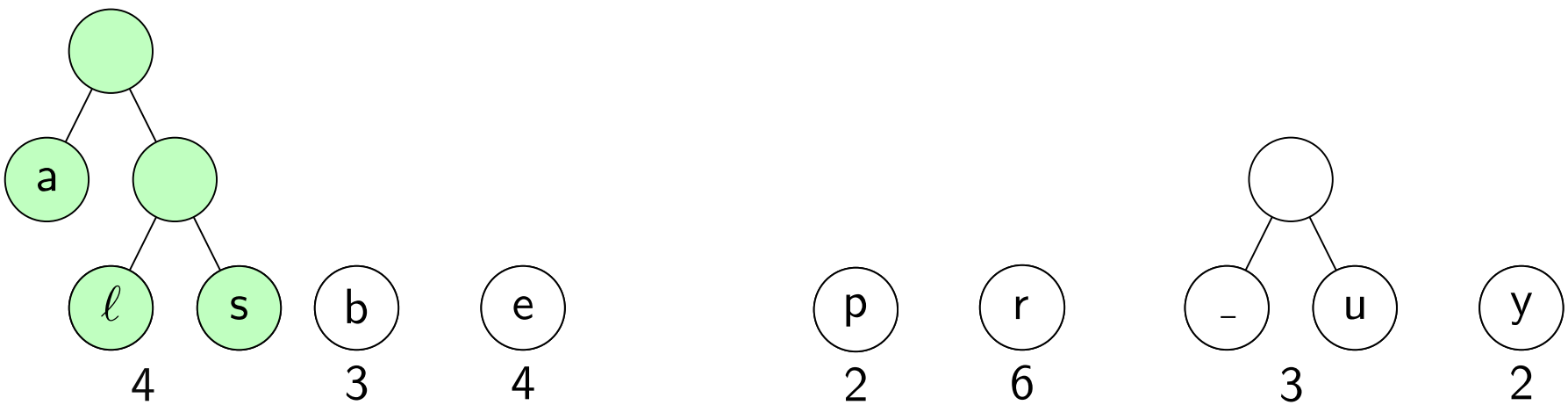
The two chosen nodes are then merged into a single tree with frequency 3.

Huffman Coding (13)



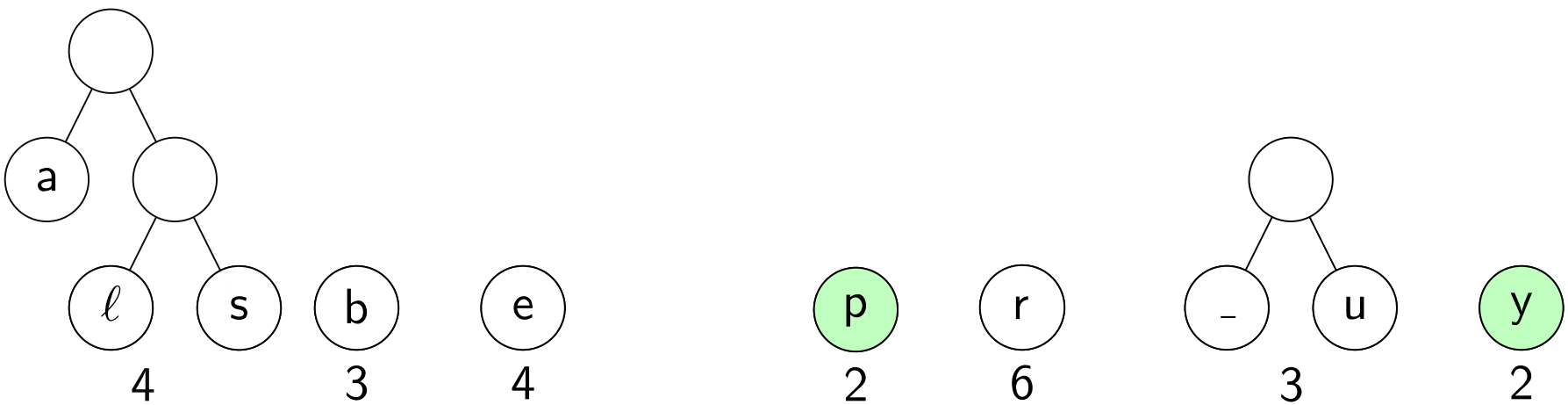
At the next step, we choose two trees with frequency 2 to merge.
In this case one of them is a previously merged tree.

Huffman Coding (14)



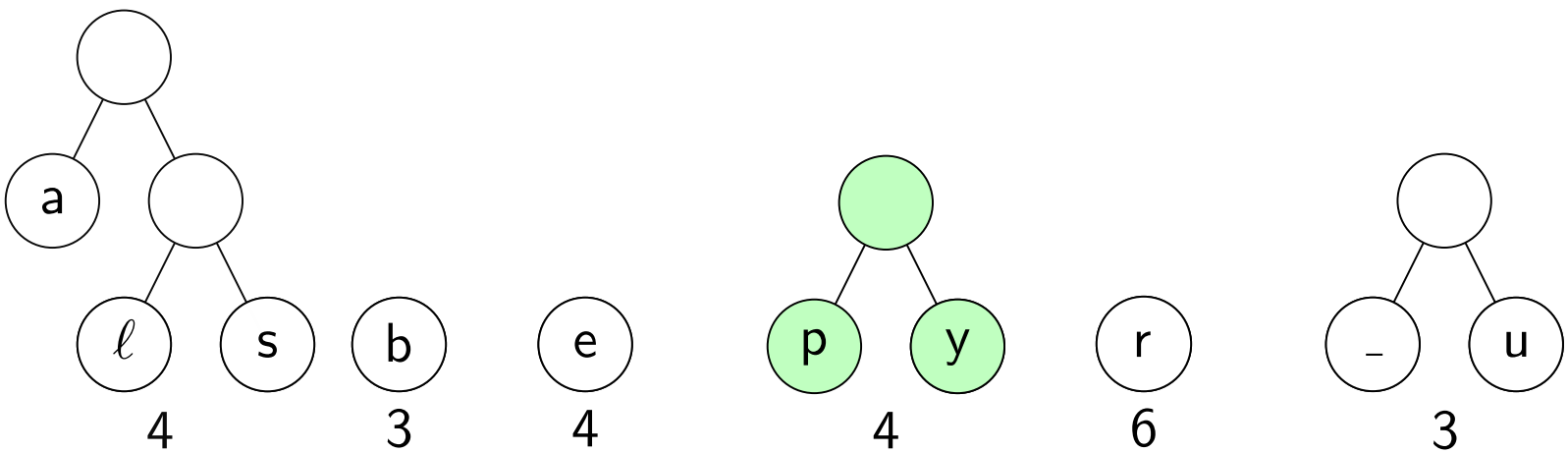
The result is a tree with frequency 4.

Huffman Coding (15)



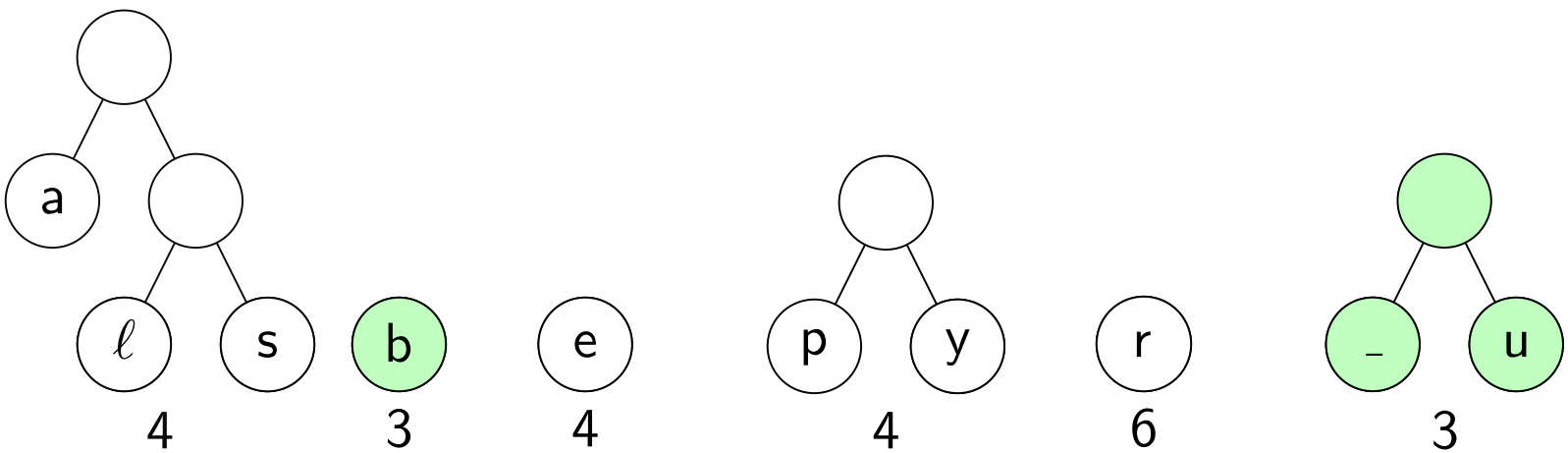
This merging process continues until only one tree remains.

Huffman Coding (16)



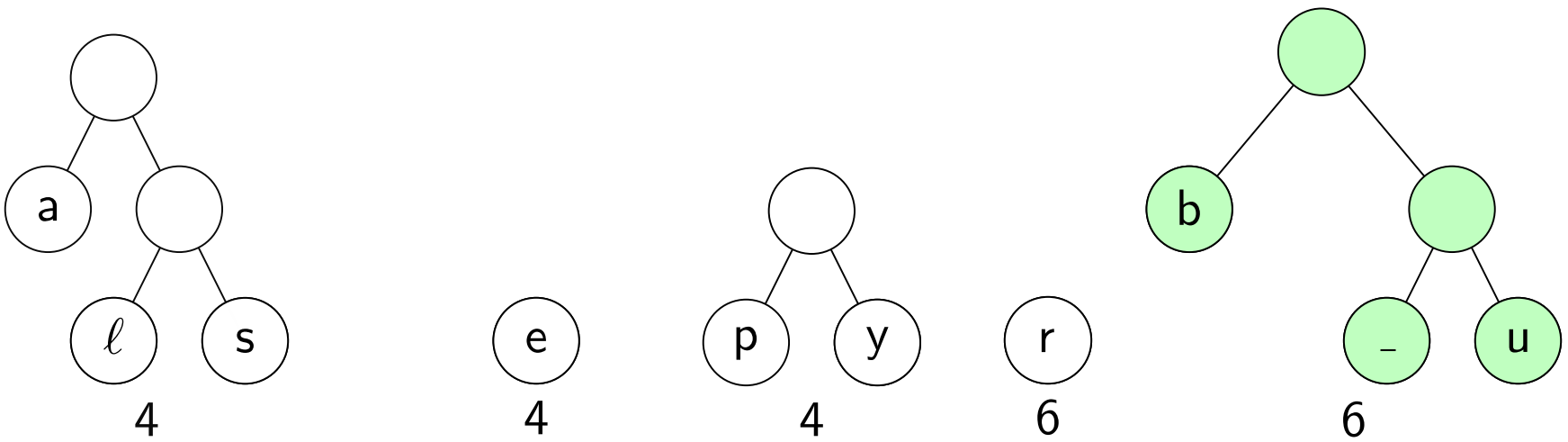
This merging process continues until only one tree remains.

Huffman Coding (17)



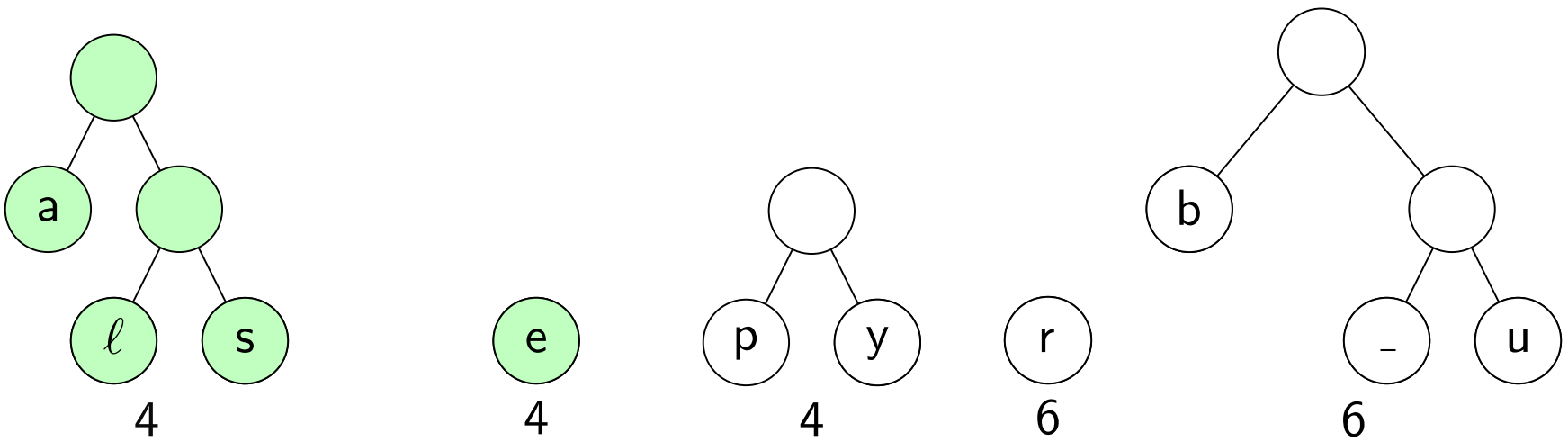
This merging process continues until only one tree remains.

Huffman Coding (18)



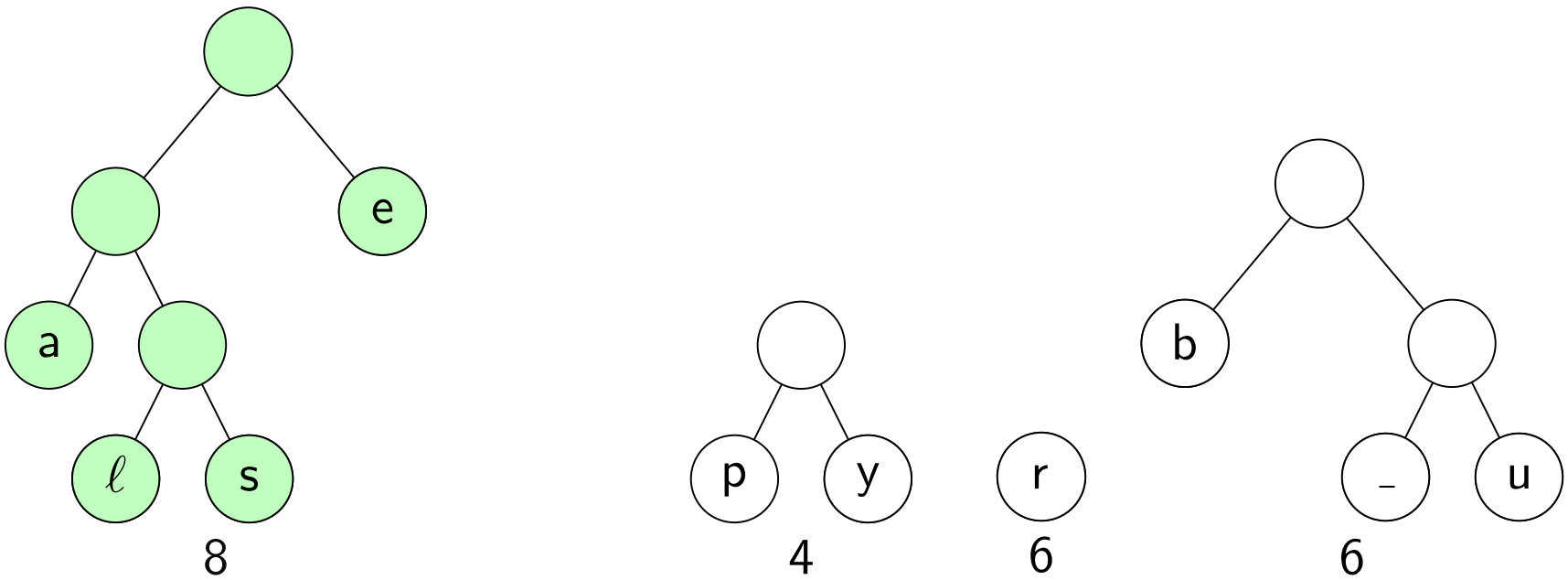
This merging process continues until only one tree remains.

Huffman Coding (19)



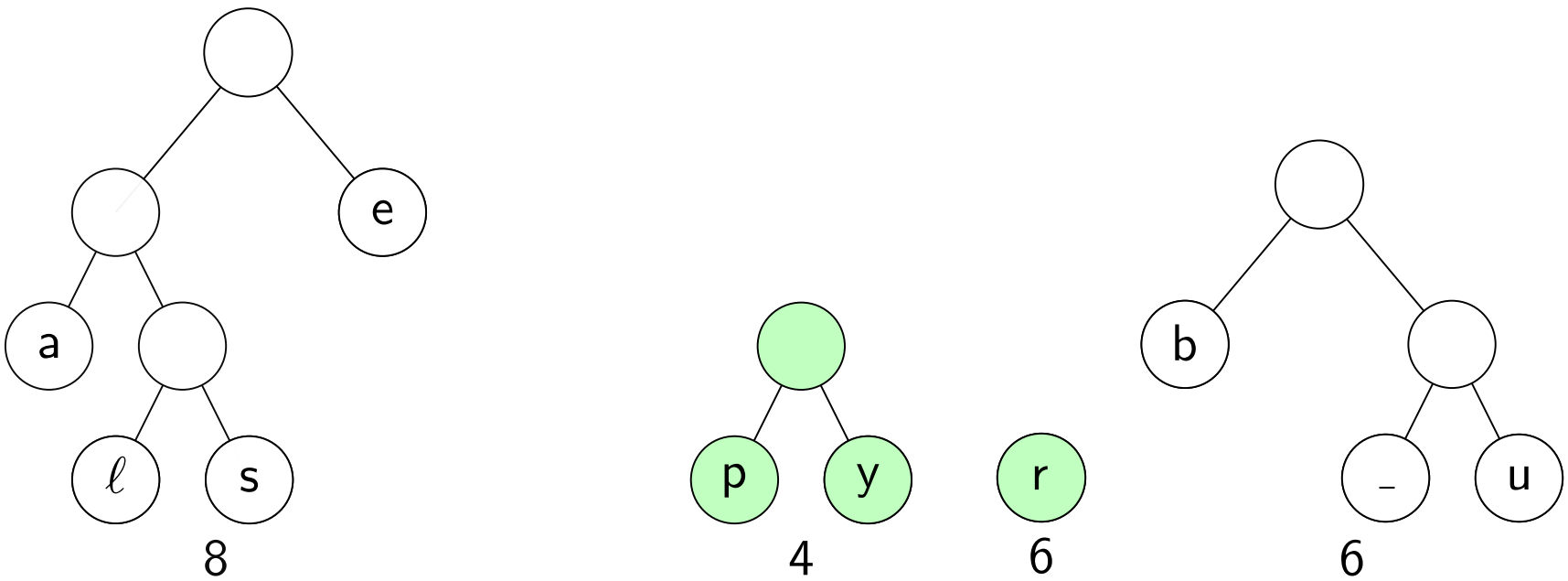
This merging process continues until only one tree remains.

Huffman Coding (20)



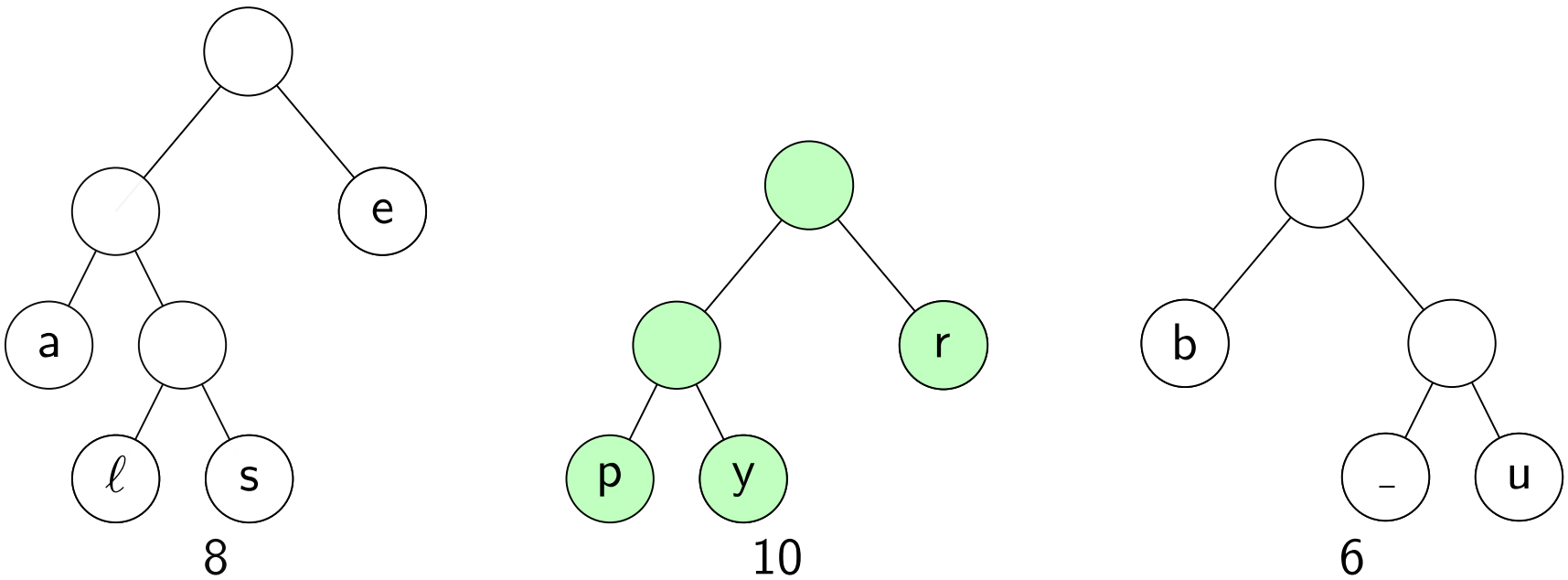
This merging process continues until only one tree remains.

Huffman Coding (21)



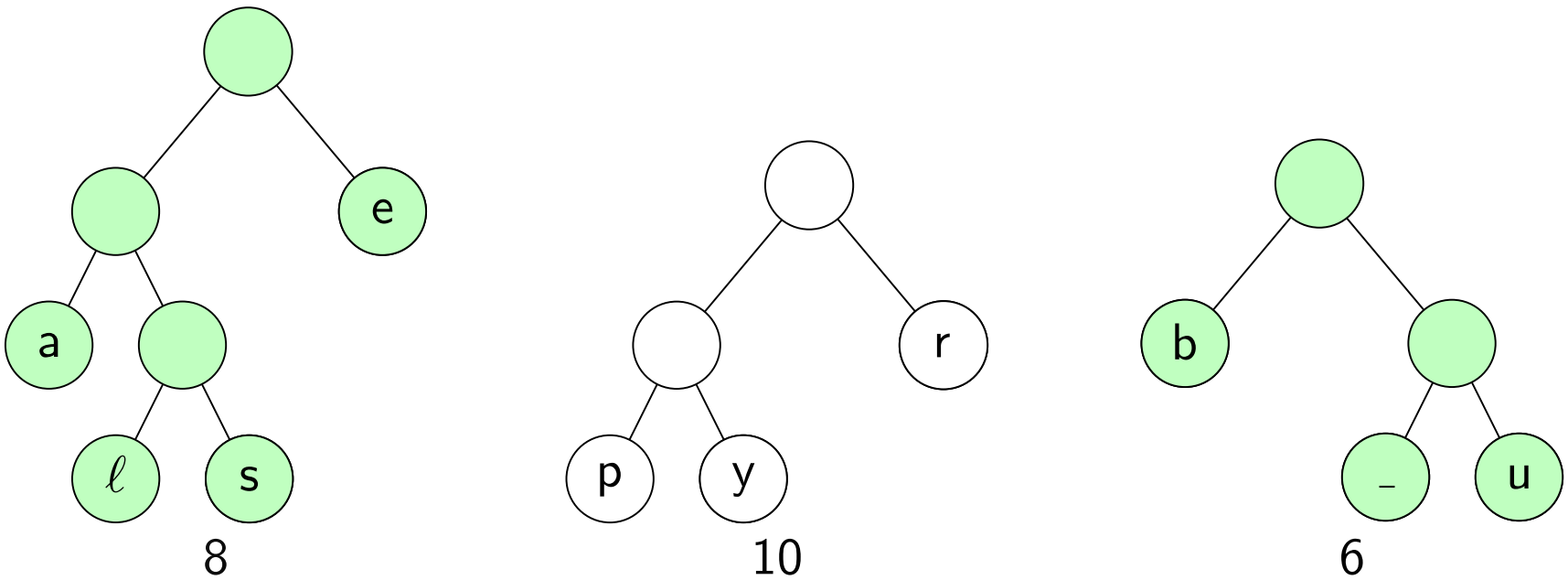
This merging process continues until only one tree remains.

Huffman Coding (22)



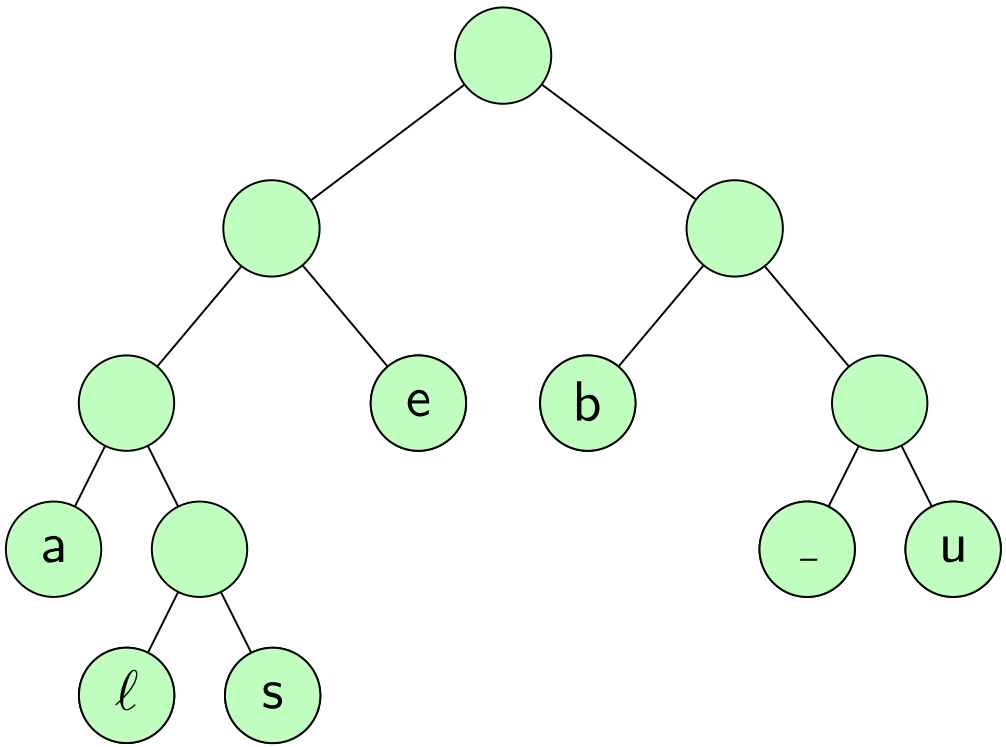
This merging process continues until only one tree remains.

Huffman Coding (23)

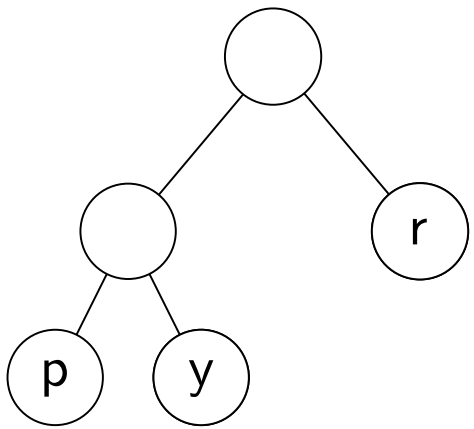


This merging process continues until only one tree remains.

Huffman Coding (24)



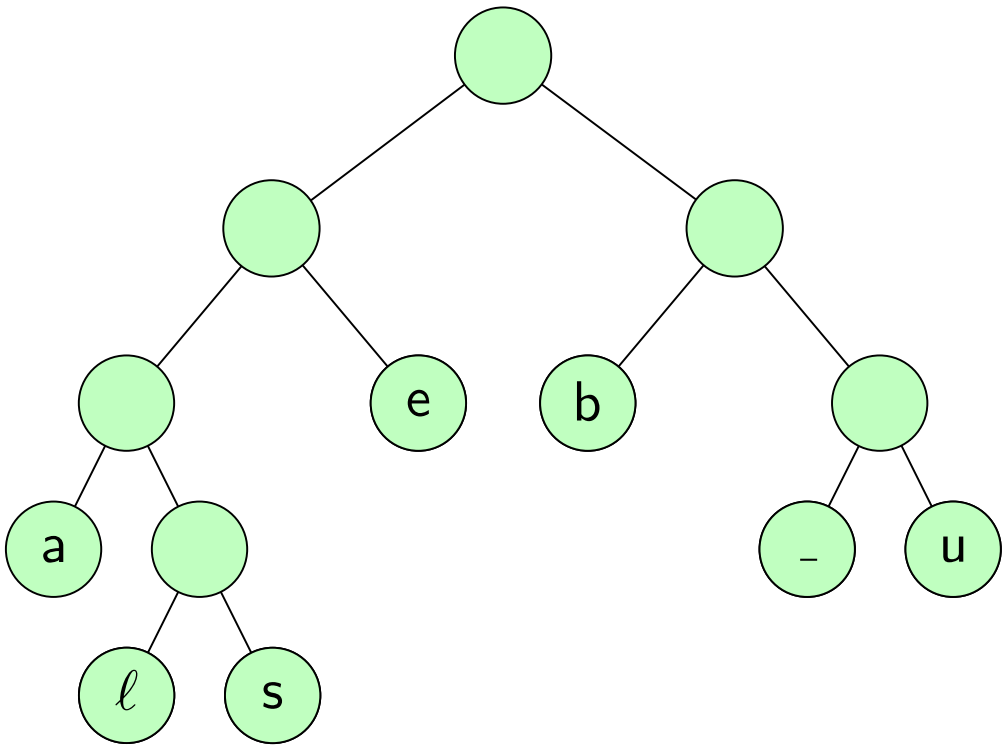
14



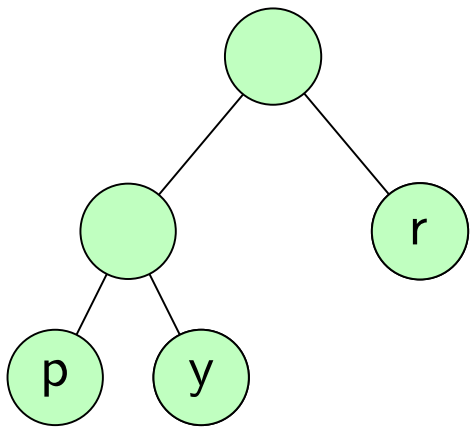
10

This merging process continues until only one tree remains.

Huffman Coding (25)



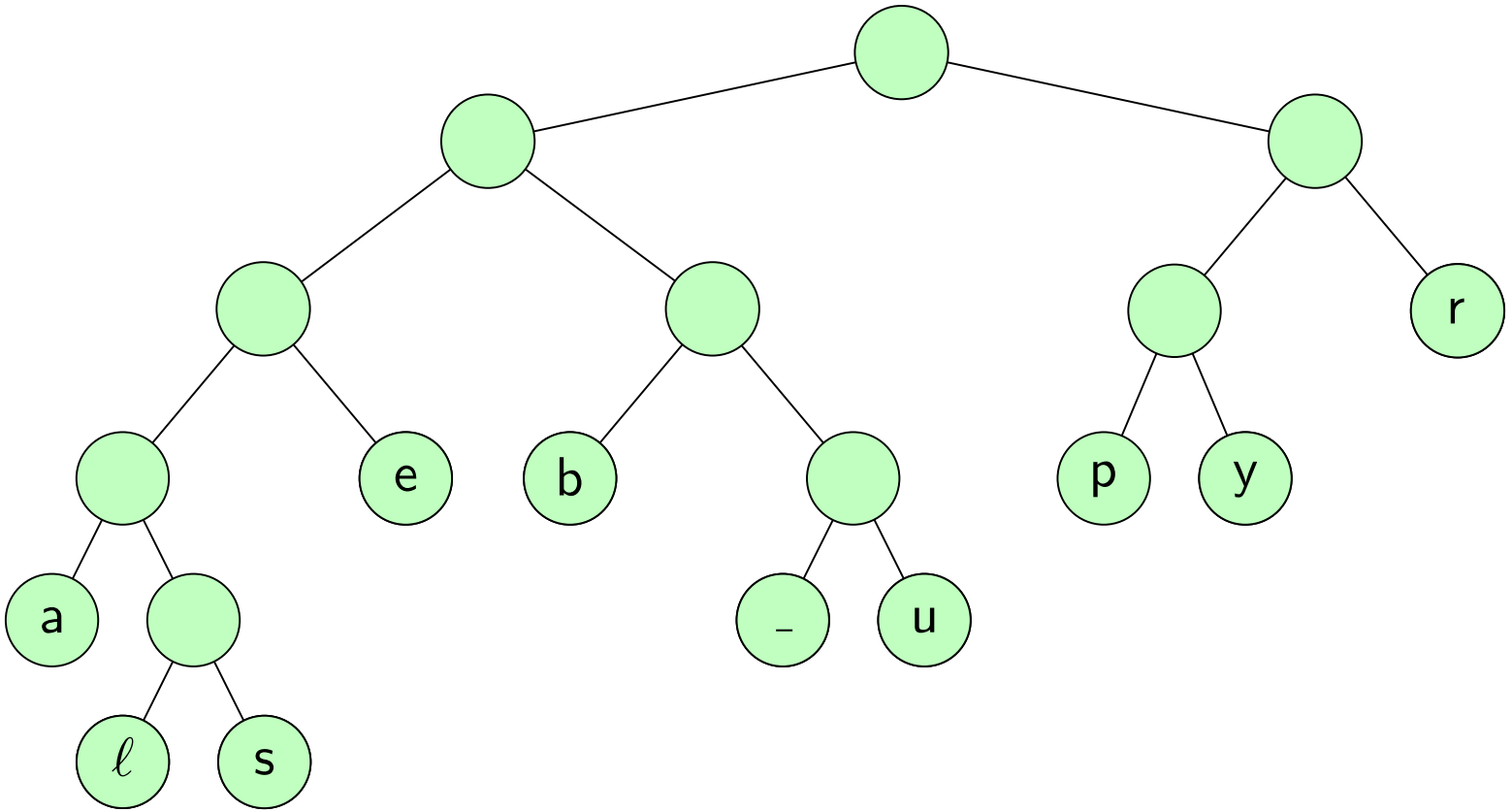
14



10

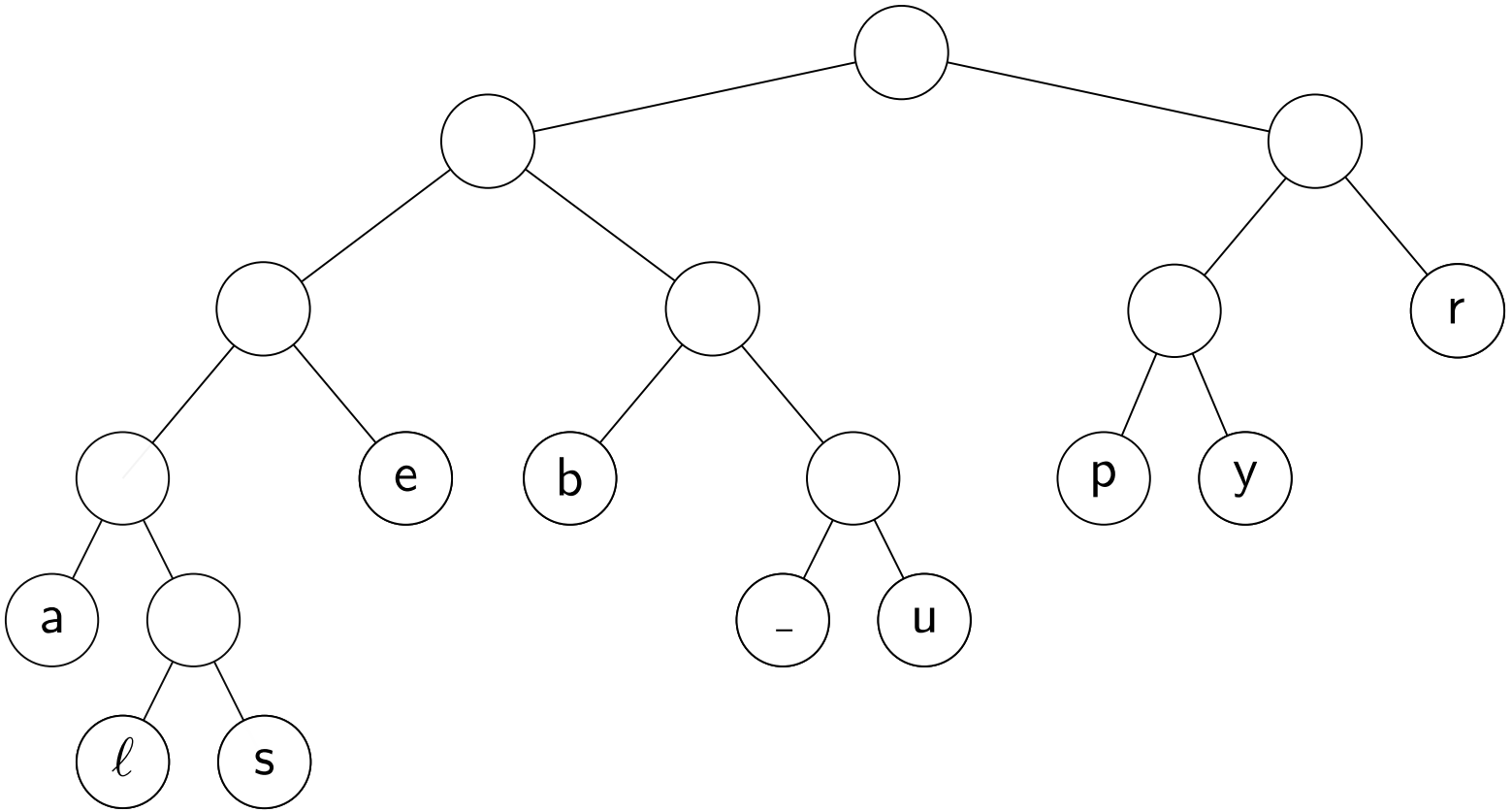
This merging process continues until only one tree remains.

Huffman Coding (26)



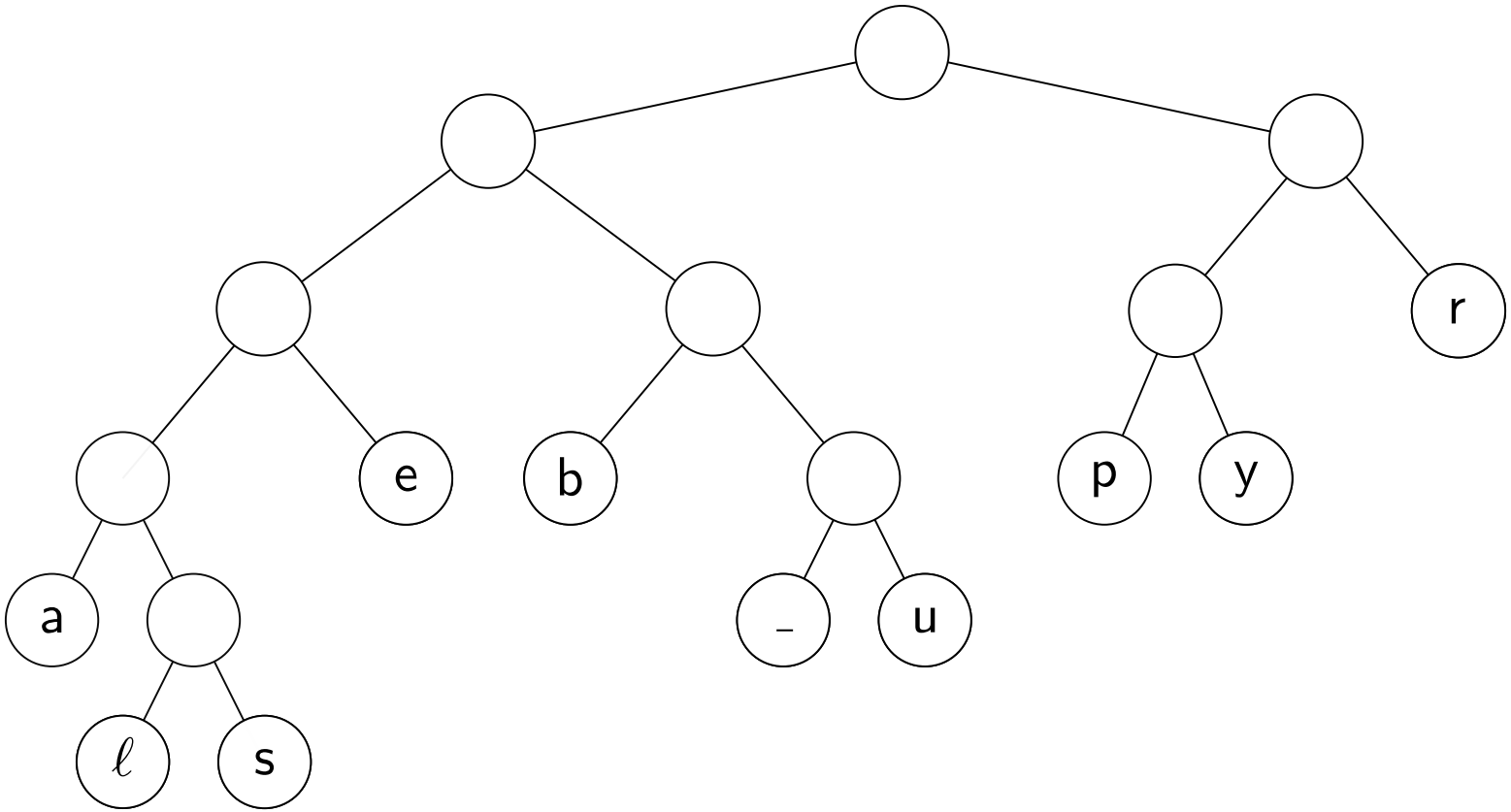
This merging process continues until only one tree remains.

Huffman Coding (27)



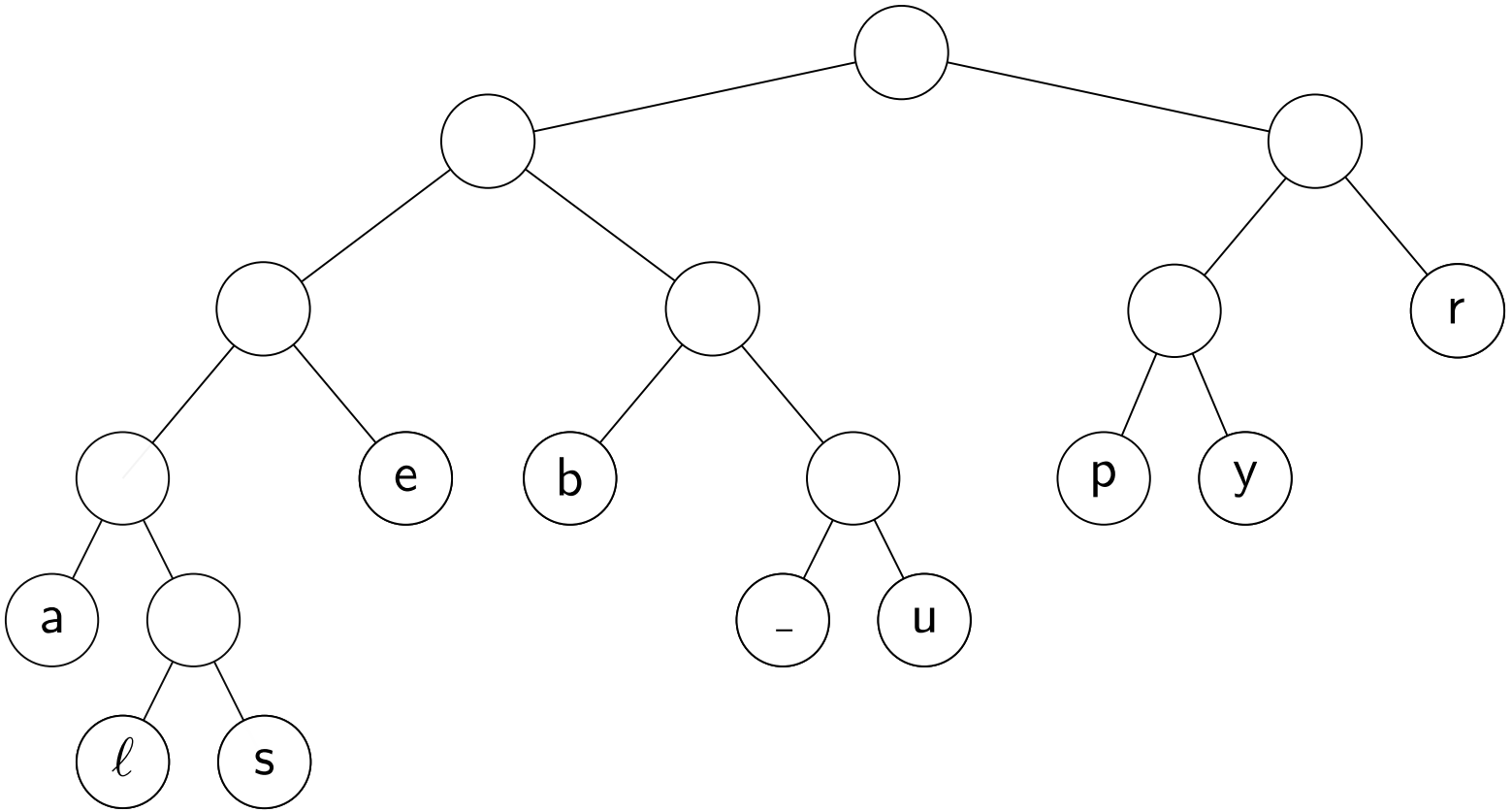
Notice that symbols which started with a low frequency were merged earlier (and likely to be involved in more merges overall), while symbols with higher frequencies were merged later.

Huffman Coding (28)



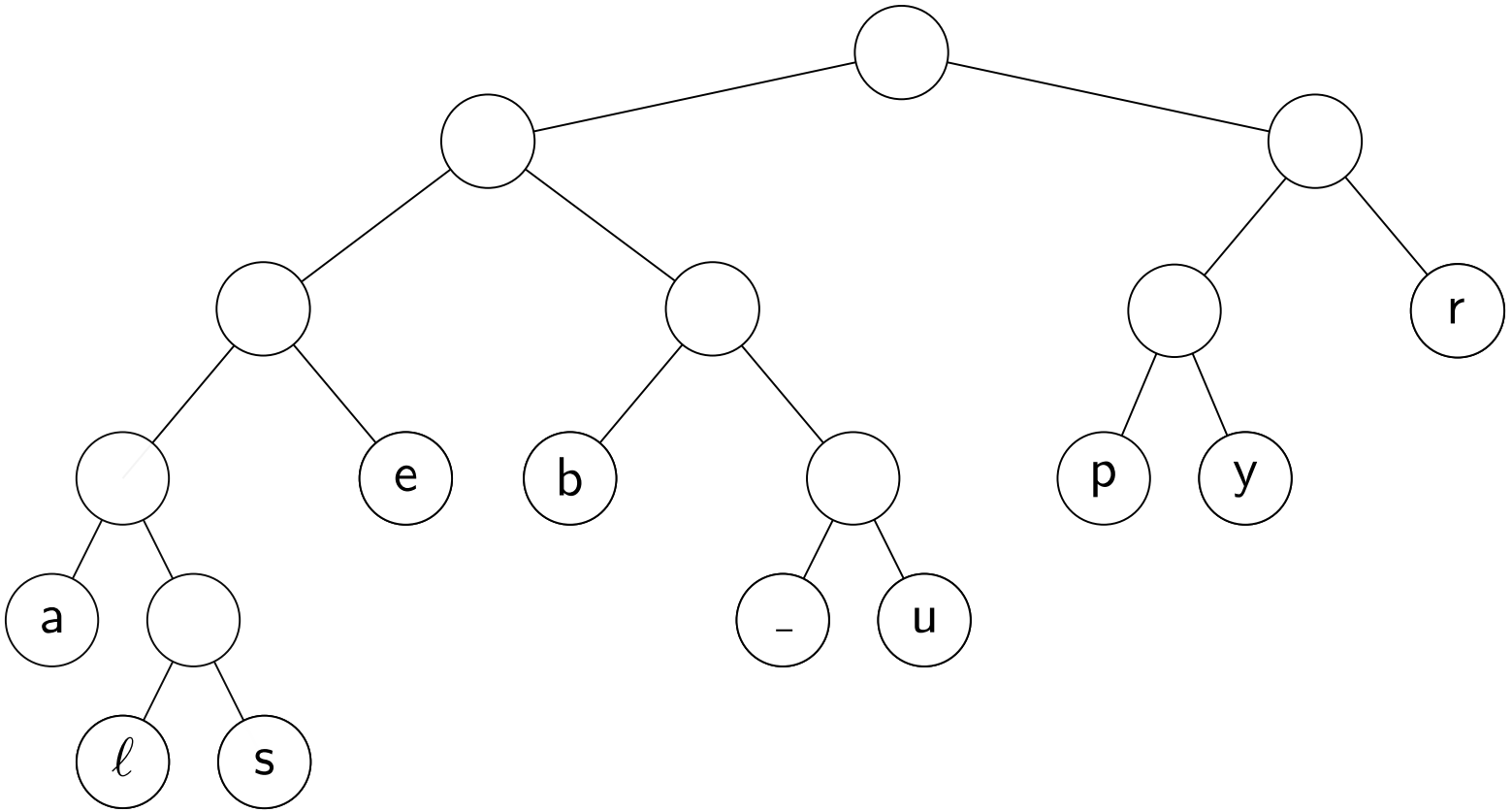
Also observe that the depth of a node in the finished tree is determined by the total number of merges involving that node.

Huffman Coding (29)



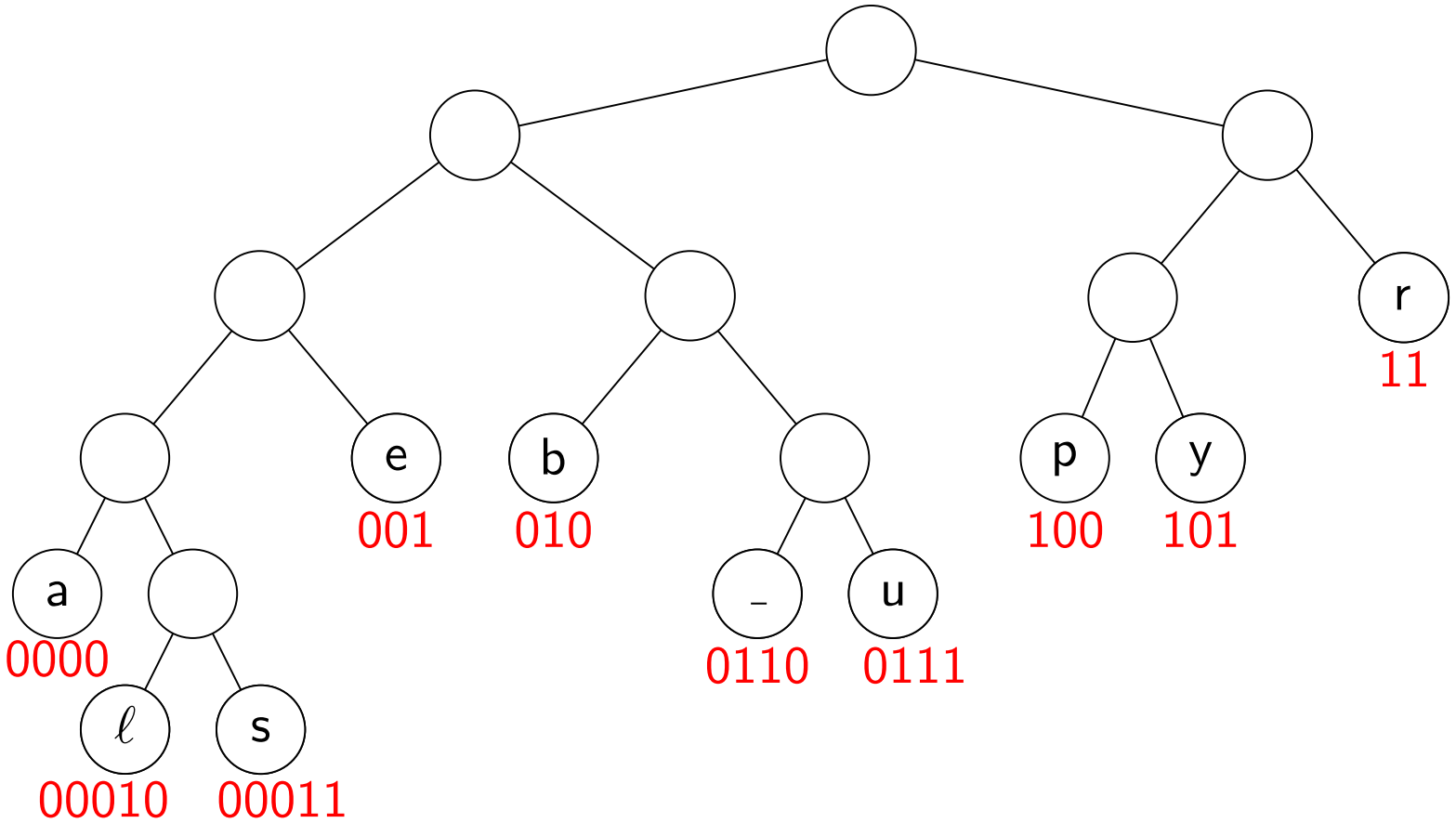
If a priority queue is used to track the frequencies of trees, each merge can be done with two RemoveMin operations and one Add operation.

Huffman Coding (30)



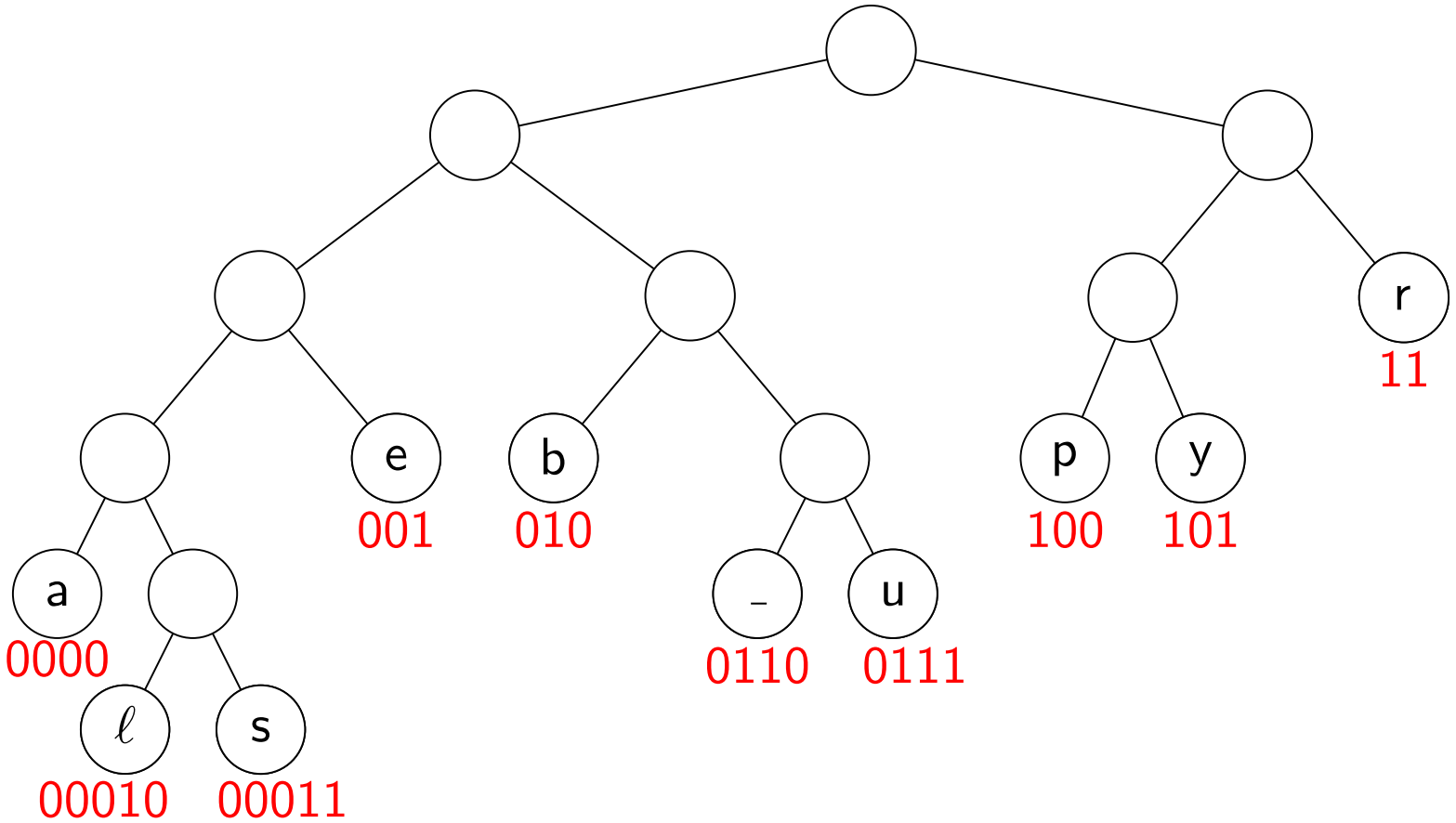
The merge itself requires constant time, so a Huffman Tree on n symbols can be constructed in $\Theta(n \log n)$ time if the frequencies are known in advance.

Huffman Coding (31)



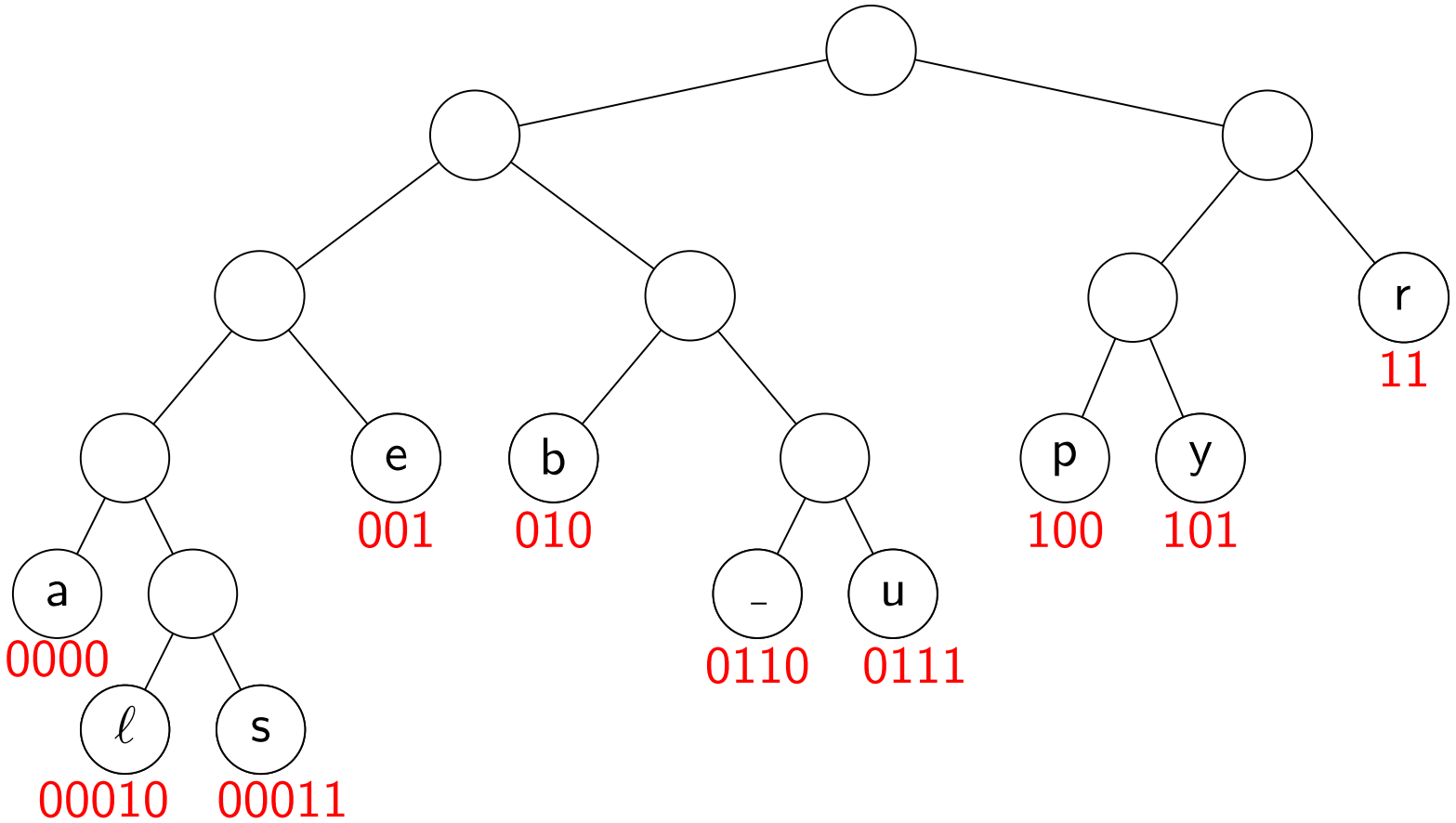
After the tree is constructed, the encoding for each symbol can be obtained by traversing the tree.

Huffman Coding (32)



A symbol table can then be constructed to make it easy to map each symbol to its encoding.

Huffman Coding (33)



_	0110	a	0000	b	010	e	001	l	00010
p	100	r	11	s	00011	u	0111	y	101

Huffman Coding (34)

Original Data

```
raspberry blueberry pear
```

24 characters, 10 distinct characters

_	0110	a	0000	b	010	e	001	l	00010
p	100	r	11	s	00011	u	0111	y	101

```
r   a    s    p   b   e   r  r   y        b   l    u   e   b   e   r
11 0000 00011 100 010 001 11 11 101 0110 010 00010 0111 001 010 001 11
r   y        p   e   a   r
11 101 0110 100 001 0000 11
```

75 bits (10 bytes)

The encoding produced by Huffman Coding is known to be optimal: the length of the compressed bit sequence is guaranteed to be the shortest possible for the set of symbols used.

Huffman Coding (35)

Original Data

```
raspberry blueberry pear
```

24 characters, 10 distinct characters

_	0110	a	0000	b	010	e	001	l	00010
p	100	r	11	s	00011	u	0111	y	101

```
r   a    s    p   b   e   r  r   y        b    l    u   e   b   e   r
11 0000 00011 100 010 001 11 11 101 0110 010 00010 0111 001 010 001 11
r   y        p   e   a   r
11 101 0110 100 001 0000 11
```

75 bits (10 bytes)

(Note, however, that the overhead involved in storing the symbol-to-encoding mapping may not be optimal)

Symbol Selection (1)

Original Data

```
raspberry blueberry cranberry pear pineapple strawberry
```

55 characters, 15 distinct characters

One character symbols

Symbol:	_	a	b	c	e	i	l	n	p	r	s	t	u	w	y
Frequency:	5	5	5	1	8	1	2	2	5	12	2	1	1	1	4

Two character symbols

Symbol:	_p	_s	an	ar	aw	be	bl	cr	ea	in
Frequency:	1	1	1	1	1	4	1	1	1	1
Symbol:	le	pe	pp	ra	rr	sp	tr	ue	y	y_
Frequency:	1	1	1	1	4	1	1	1	1	3

Once the symbol set is chosen, Huffman Coding will always produce an optimal encoding. However, choosing a good set of symbols is not trivial.

Symbol Selection (2)

Original Data

```
raspberry blueberry cranberry pear pineapple strawberry
```

55 characters, 15 distinct characters

One character symbols

Symbol:	_	a	b	c	e	i	l	n	p	r	s	t	u	w	y
Frequency:	5	5	5	1	8	1	2	2	5	12	2	1	1	1	4

Two character symbols

Symbol:	_p	_s	an	ar	aw	be	bl	cr	ea	in
Frequency:	1	1	1	1	1	4	1	1	1	1
Symbol:	le	pe	pp	ra	rr	sp	tr	ue	y	y_
Frequency:	1	1	1	1	4	1	1	1	1	3

Using single characters is certainly convenient, but using pairs of characters (or even mixed groups, like ‘r’ and ’berry’) might result in better compression, by leveraging patterns in the input data.

Decoding Prefix-Free Codes (1)

Question: Given a symbol table (mapping symbols to encodings) for a prefix-free code and a sequence of compressed bits, how can we efficiently decompress the data?

Decoding Prefix-Free Codes (2)

Compressed Data

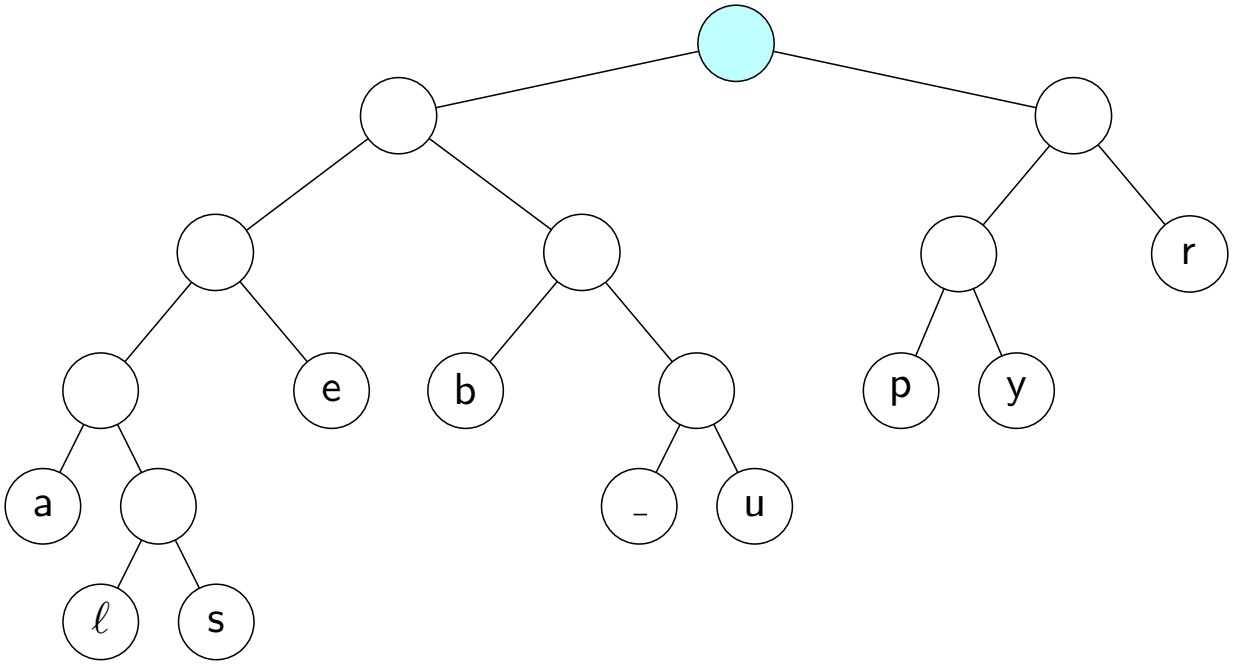
```
1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 1 0 0 0 1 1 1 1 1 1 0 1 0 1
1 0 0 1 0 0 0 0 1 0 0 1 1 1 0 0 1 0 1 0 0 0 1 1 1 1 1 1 0
1 0 1 1 0 1 0 0 0 0 1 0 0 0 0 1 1
```

Symbol Table

_	0110	a	0000	b	010	e	001	l	00010
p	100	r	11	s	00011	u	0111	y	101

Although we could try using a dictionary to map bit encodings to symbols, it would be difficult to know how many bits should be used in a lookup operation. Instead, the best approach is to rebuild the encoding tree from the symbol table.

Decoding Prefix-Free Codes (3)



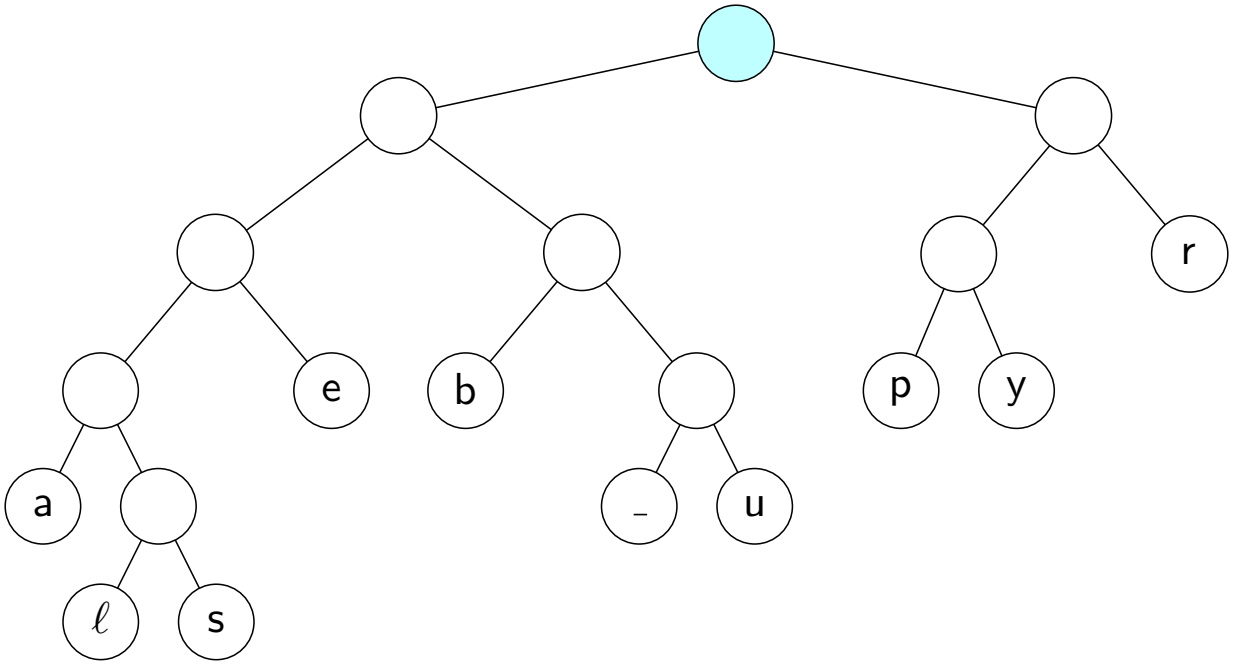
Compressed Data

```
1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 1 0 0 0 1 1 1 1 1 1 0 1 0 1
1 0 0 1 0 0 0 0 1 0 0 1 1 1 0 0 1 0 1 0 0 0 1 1 1 1 1 1 0
1 0 1 1 0 1 0 0 0 0 1 0 0 0 0 1 1
```

Decompressed Data:

To decompress, we use the compressed bits as directions in a walk through the tree.

Decoding Prefix-Free Codes (4)



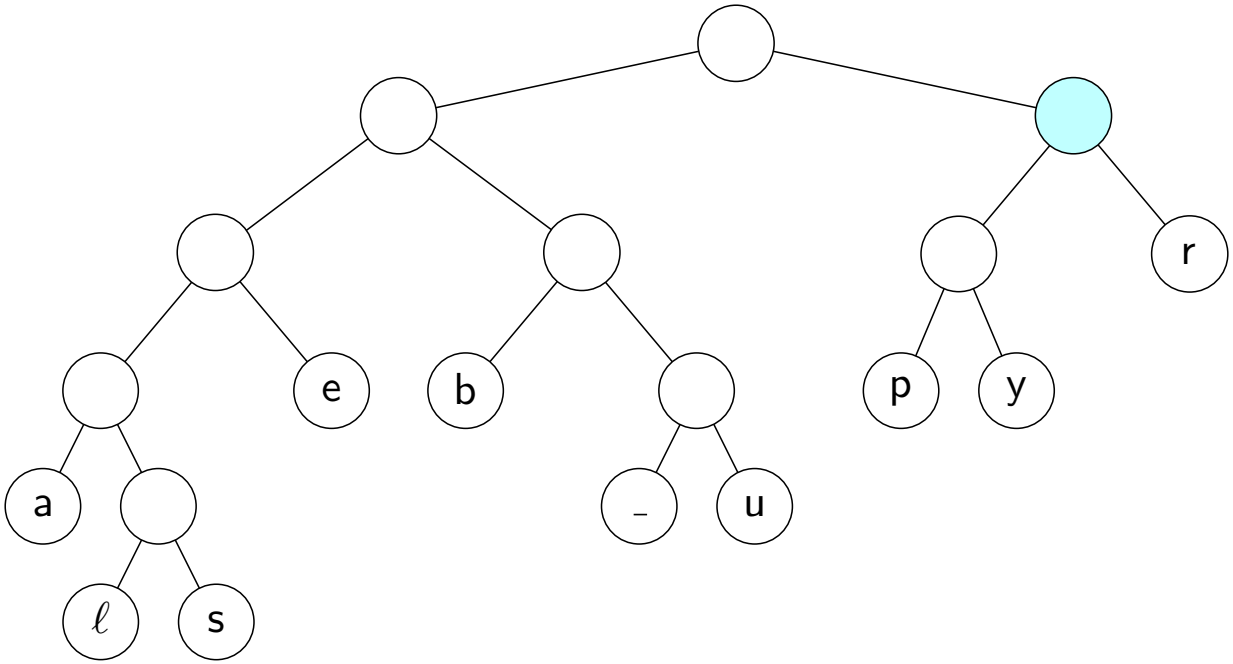
Compressed Data

```
1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 1 0 0 0 1 1 1 1 1 1 0 1 0 1
1 0 0 1 0 0 0 0 1 0 0 1 1 1 0 0 1 0 1 0 0 0 1 1 1 1 1 1 0
1 0 1 1 0 1 0 0 0 0 1 0 0 0 0 1 1
```

Decompressed Data:

At each step, the walk will be positioned at one node (highlighted in green), starting at the root.

Decoding Prefix-Free Codes (5)



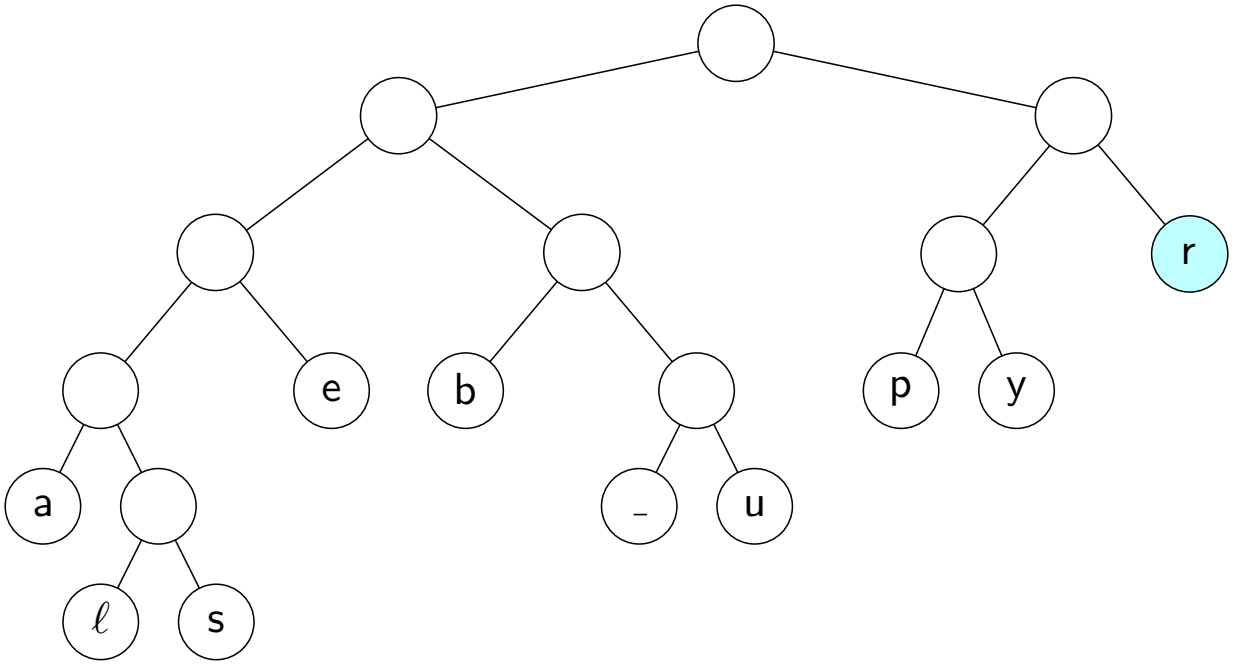
Compressed Data

```
1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 1 0 0 0 1 1 1 1 1 1 0 1 0 1
1 0 0 1 0 0 0 0 1 0 0 1 1 1 0 0 1 0 1 0 0 0 1 1 1 1 1 1 0
1 0 1 1 0 1 0 0 0 0 1 0 0 0 0 1 1
```

Decompressed Data:

As each compressed bit is read, the walk moves either to the left child (a 0 bit) or the right child (a 1 bit).

Decoding Prefix-Free Codes (6)



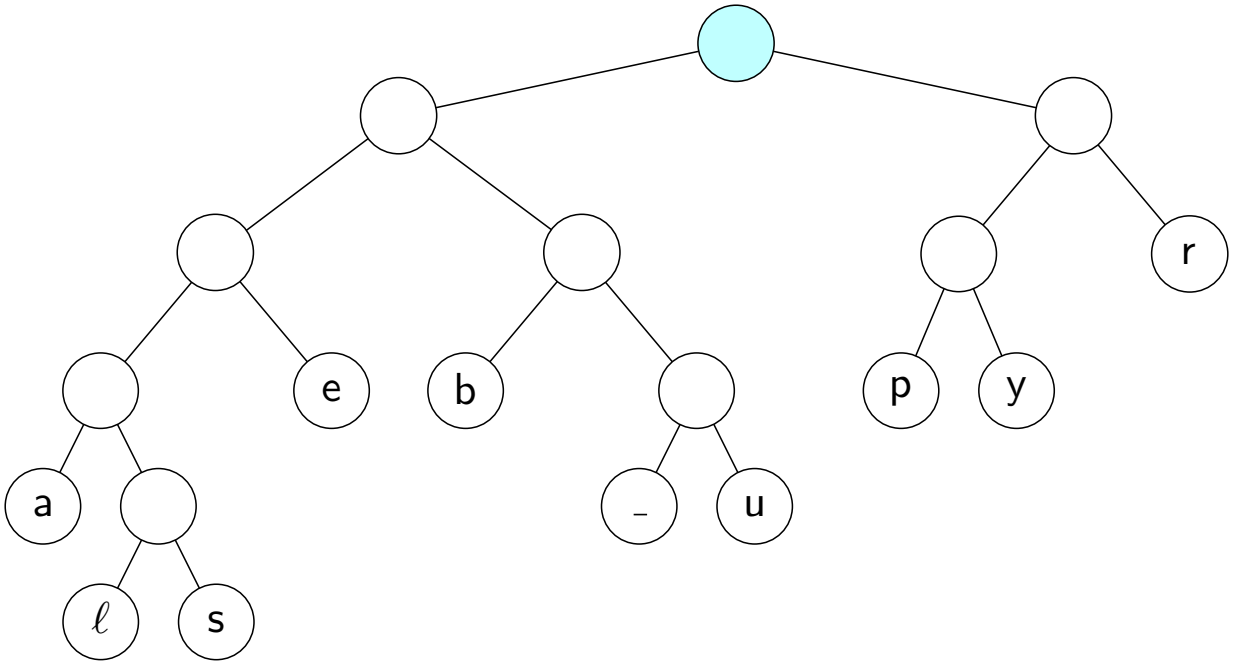
Compressed Data

```
1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 1 0 0 0 1 1 1 1 1 1 0 1 0 1
1 0 0 1 0 0 0 0 1 0 0 1 1 1 0 0 1 0 1 0 0 0 1 1 1 1 1 1 0
1 0 1 1 0 1 0 0 0 0 1 0 0 0 0 1 1
```

Decompressed Data: r

If the walk arrives at a leaf, the symbol in that leaf is output.

Decoding Prefix-Free Codes (7)



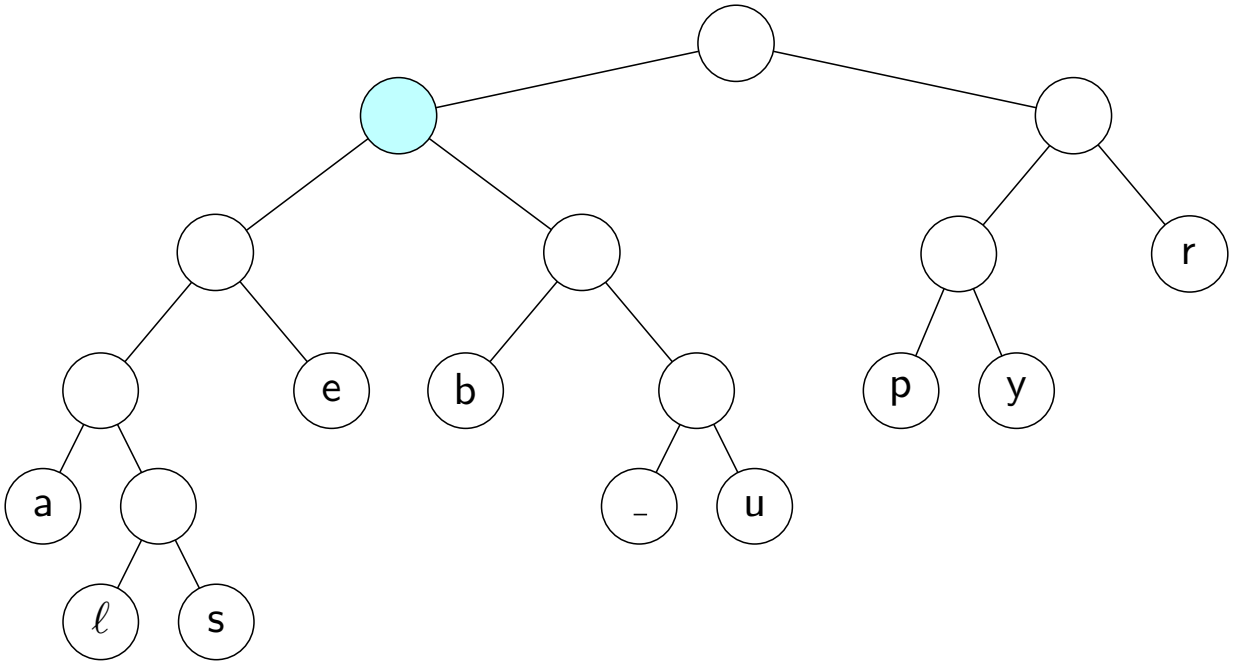
Compressed Data

```
1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 1 0 0 0 1 1 1 1 1 1 0 1 0 1
1 0 0 1 0 0 0 0 1 0 0 1 1 1 0 0 1 0 1 0 0 0 1 1 1 1 1 1 0
1 0 1 1 0 1 0 0 0 0 1 0 0 0 0 1 1
```

Decompressed Data: `r`

After the symbol is output, the walk returns to the root before reading the next compressed bit.

Decoding Prefix-Free Codes (8)



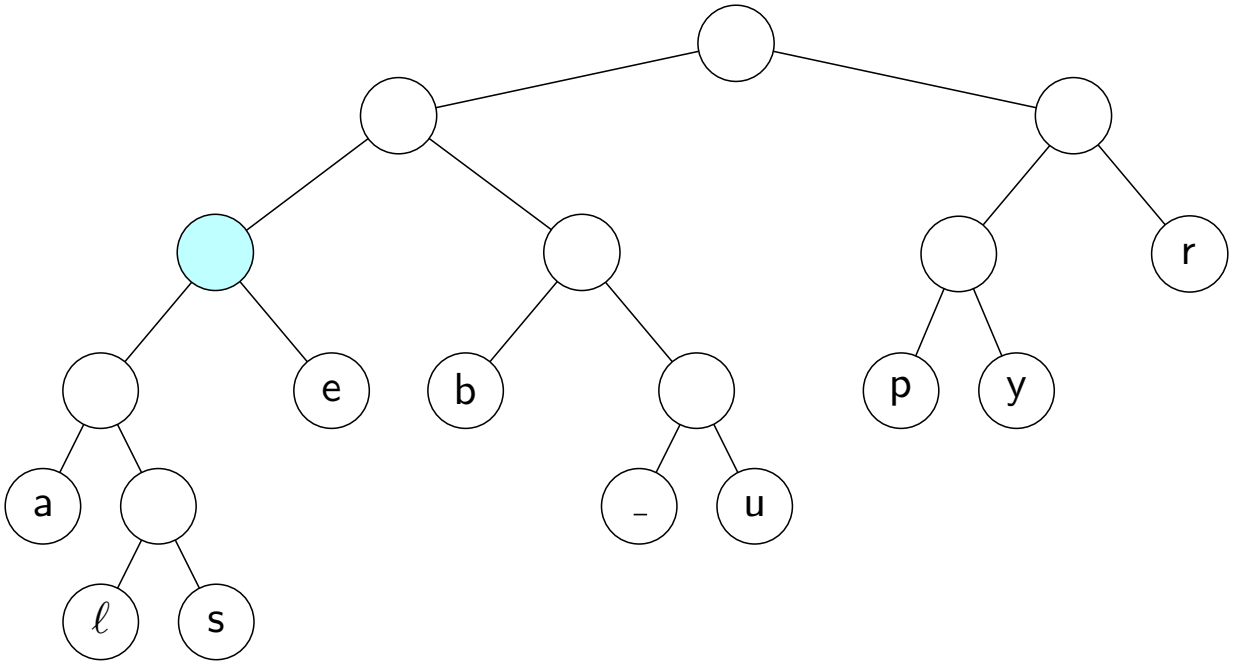
Compressed Data

```
1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 1 0 0 0 1 1 1 1 1 1 0 1 0 1
1 0 0 1 0 0 0 0 1 0 0 1 1 1 0 0 1 0 1 0 0 0 1 1 1 1 1 1 0
1 0 1 1 0 1 0 0 0 0 1 0 0 0 0 1 1
```

Decompressed Data: `r`

Each step requires constant time, so the entire process is $\Theta(n)$ on n bits of compressed data.

Decoding Prefix-Free Codes (9)



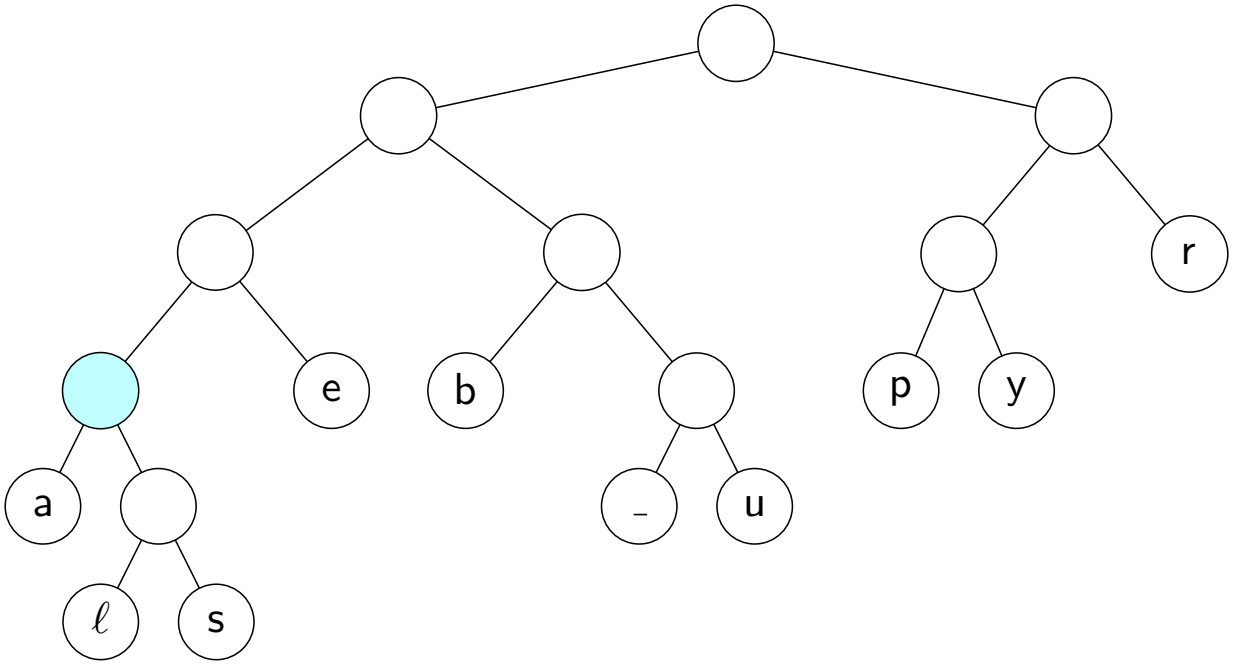
Compressed Data

```
1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 1 0 0 0 1 1 1 1 1 1 0 1 0 1
1 0 0 1 0 0 0 0 1 0 0 1 1 1 0 0 1 0 1 0 0 0 1 1 1 1 1 1 0
1 0 1 1 0 1 0 0 0 0 1 0 0 0 0 1 1
```

Decompressed Data: `r`

Each step requires constant time, so the entire process is $\Theta(n)$ on n bits of compressed data.

Decoding Prefix-Free Codes (10)



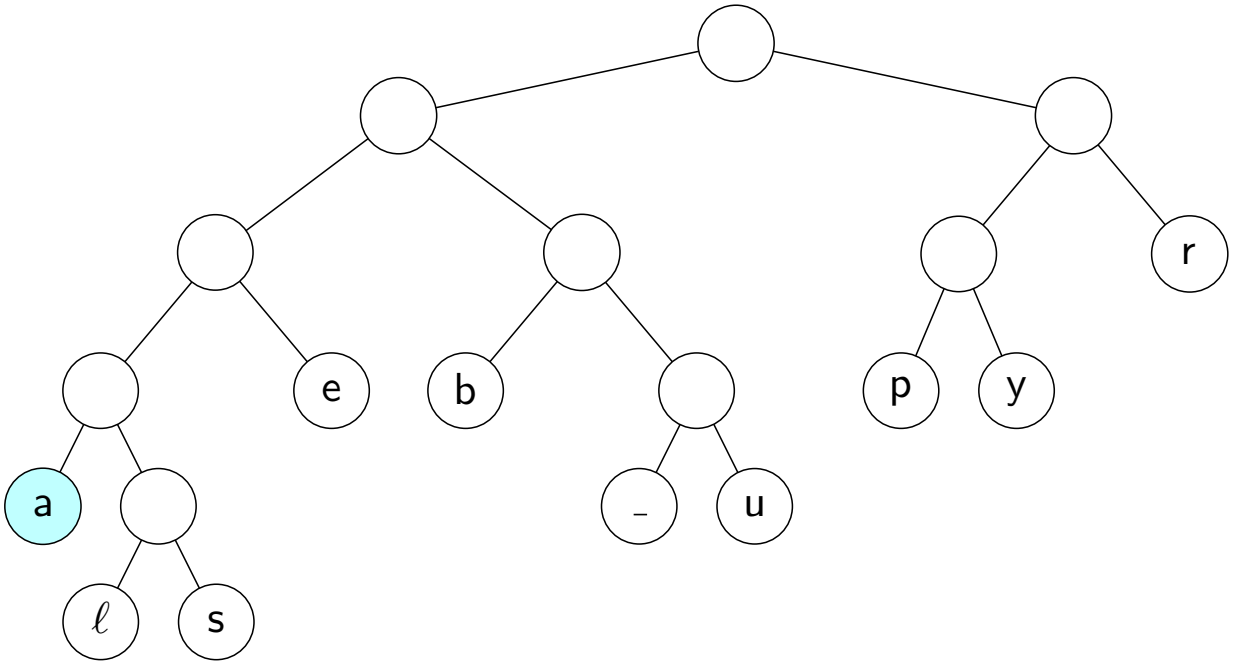
Compressed Data

```
1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 1 0 0 0 1 1 1 1 1 1 0 1 0 1
1 0 0 1 0 0 0 0 1 0 0 1 1 1 0 0 1 0 1 0 0 0 1 1 1 1 1 1 0
1 0 1 1 0 1 0 0 0 0 1 0 0 0 0 1 1
```

Decompressed Data: `r`

Each step requires constant time, so the entire process is $\Theta(n)$ on n bits of compressed data.

Decoding Prefix-Free Codes (11)



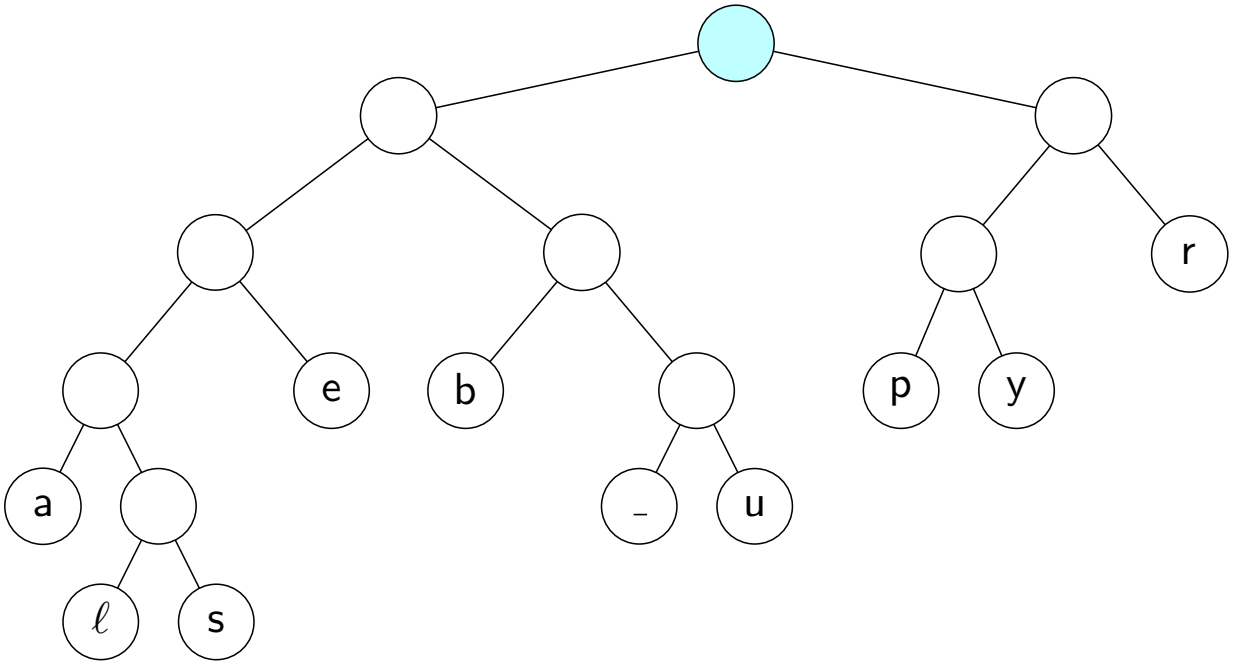
Compressed Data

```
1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 1 0 0 0 1 1 1 1 1 1 0 1 0 1
1 0 0 1 0 0 0 0 1 0 0 1 1 1 0 0 1 0 1 0 0 0 1 1 1 1 1 1 0
1 0 1 1 0 1 0 0 0 0 1 0 0 0 0 1 1
```

Decompressed Data: `ra`

Each step requires constant time, so the entire process is $\Theta(n)$ on n bits of compressed data.

Decoding Prefix-Free Codes (12)



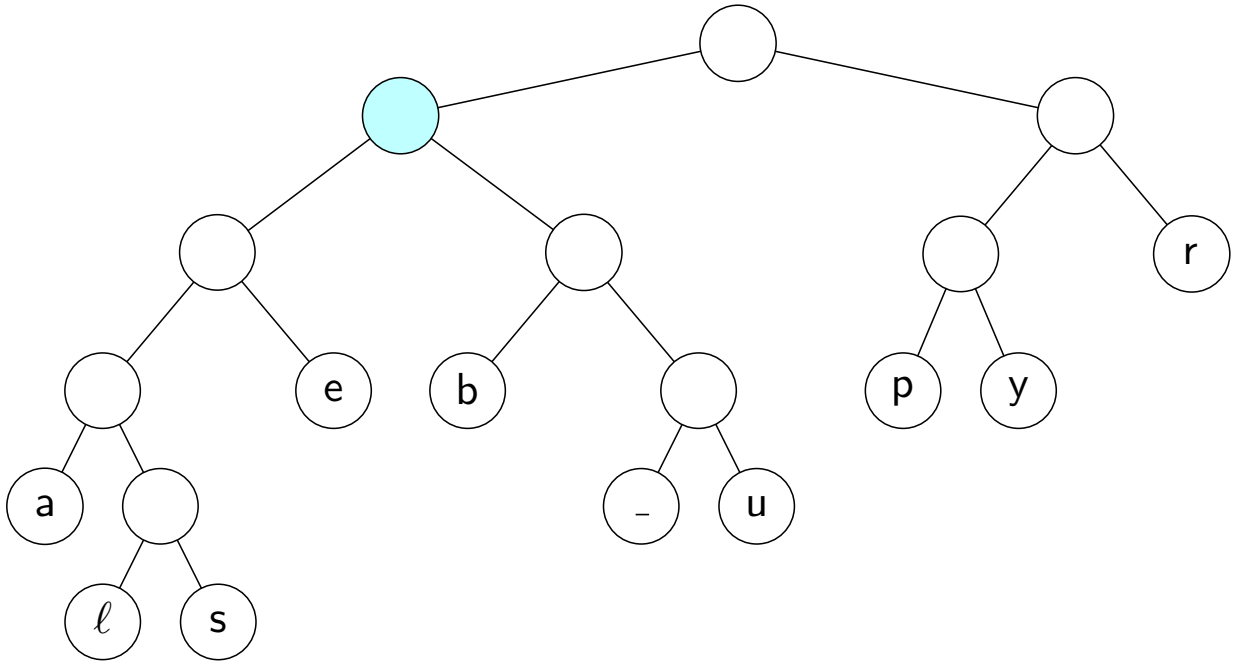
Compressed Data

```
1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 1 0 0 0 1 1 1 1 1 1 0 1 0 1
1 0 0 1 0 0 0 0 1 0 0 1 1 1 0 0 1 0 1 0 0 0 1 1 1 1 1 1 0
1 0 1 1 0 1 0 0 0 0 1 0 0 0 0 1 1
```

Decompressed Data: `ra`

Each step requires constant time, so the entire process is $\Theta(n)$ on n bits of compressed data.

Decoding Prefix-Free Codes (13)



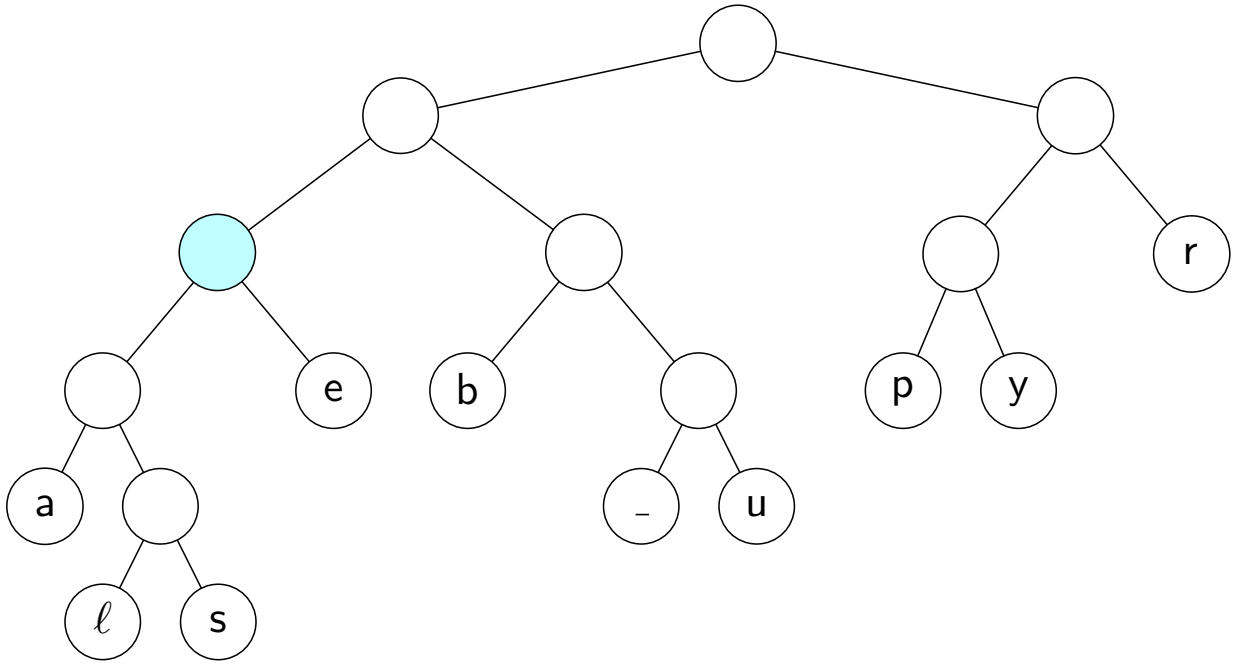
Compressed Data

```
1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 1 0 0 0 1 1 1 1 1 1 0 1 0 1
1 0 0 1 0 0 0 0 1 0 0 1 1 1 0 0 1 0 1 0 0 0 1 1 1 1 1 1 0
1 0 1 1 0 1 0 0 0 0 1 0 0 0 0 1 1
```

Decompressed Data: `ra`

Each step requires constant time, so the entire process is $\Theta(n)$ on n bits of compressed data.

Decoding Prefix-Free Codes (14)



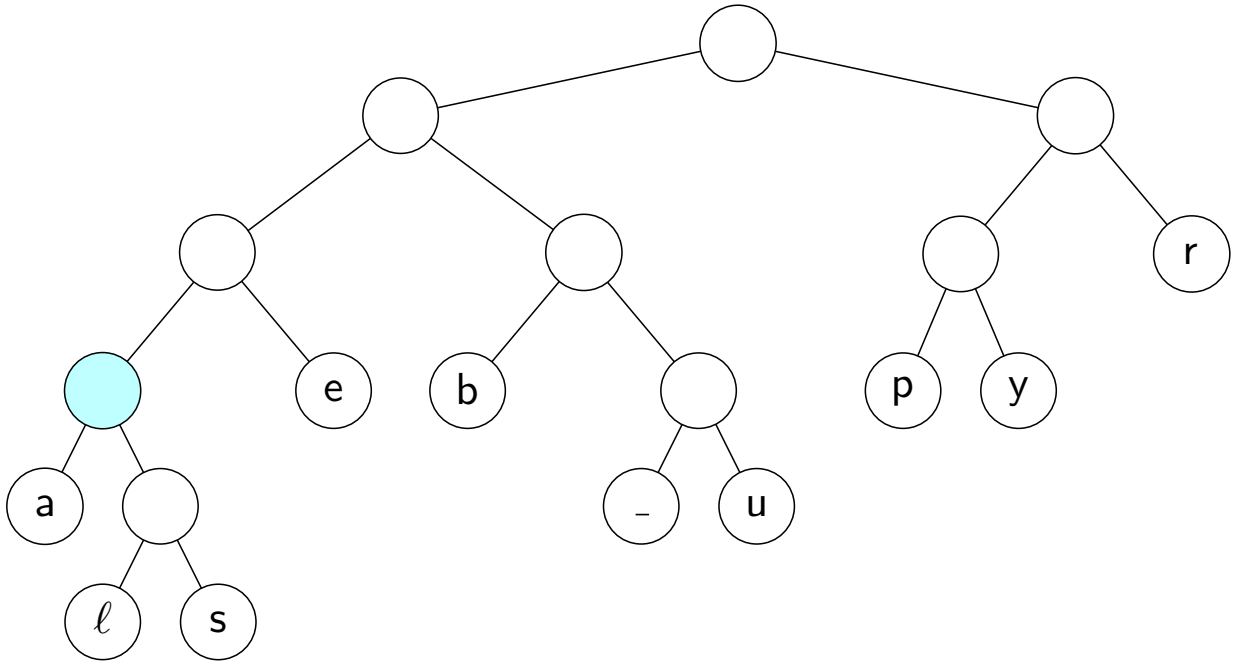
Compressed Data

```
1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 1 0 0 0 1 1 1 1 1 1 0 1 0 1
1 0 0 1 0 0 0 0 1 0 0 1 1 1 0 0 1 0 1 0 0 0 1 1 1 1 1 1 0
1 0 1 1 0 1 0 0 0 0 1 0 0 0 0 1 1
```

Decompressed Data: `ra`

Each step requires constant time, so the entire process is $\Theta(n)$ on n bits of compressed data.

Decoding Prefix-Free Codes (15)



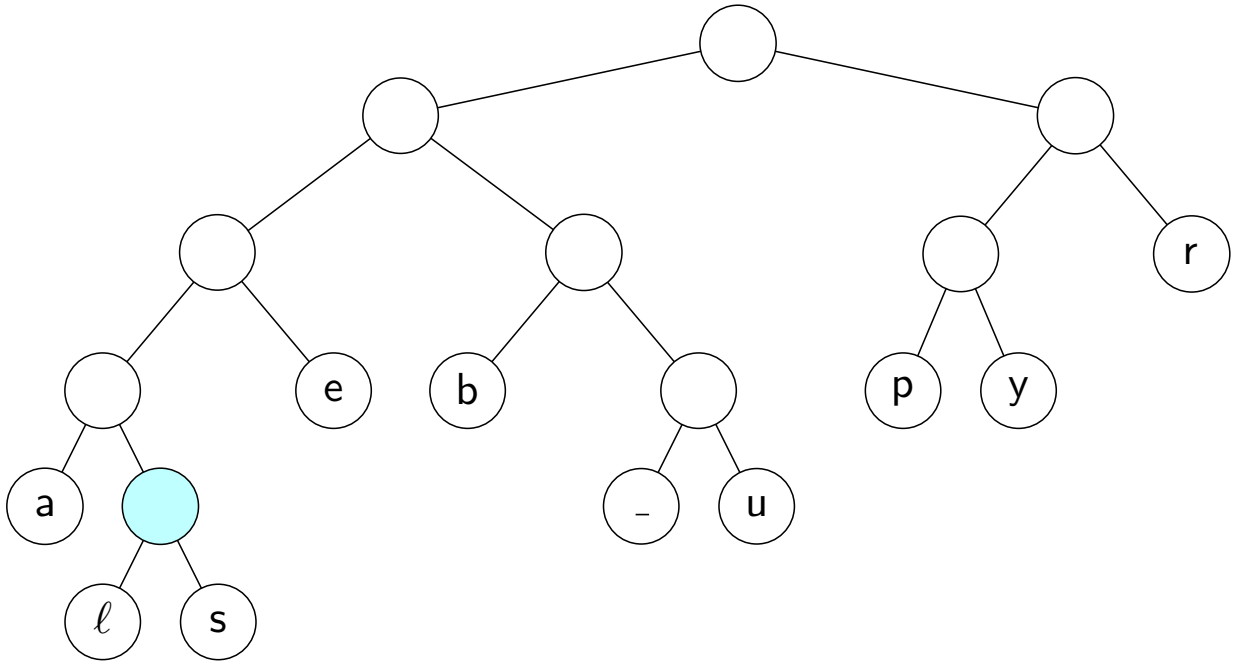
Compressed Data

```
1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 1 0 0 0 1 1 1 1 1 1 0 1 0 1
1 0 0 1 0 0 0 0 1 0 0 1 1 1 0 0 1 0 1 0 0 0 1 1 1 1 1 1 0
1 0 1 1 0 1 0 0 0 0 1 0 0 0 0 1 1
```

Decompressed Data: `ra`

Each step requires constant time, so the entire process is $\Theta(n)$ on n bits of compressed data.

Decoding Prefix-Free Codes (16)



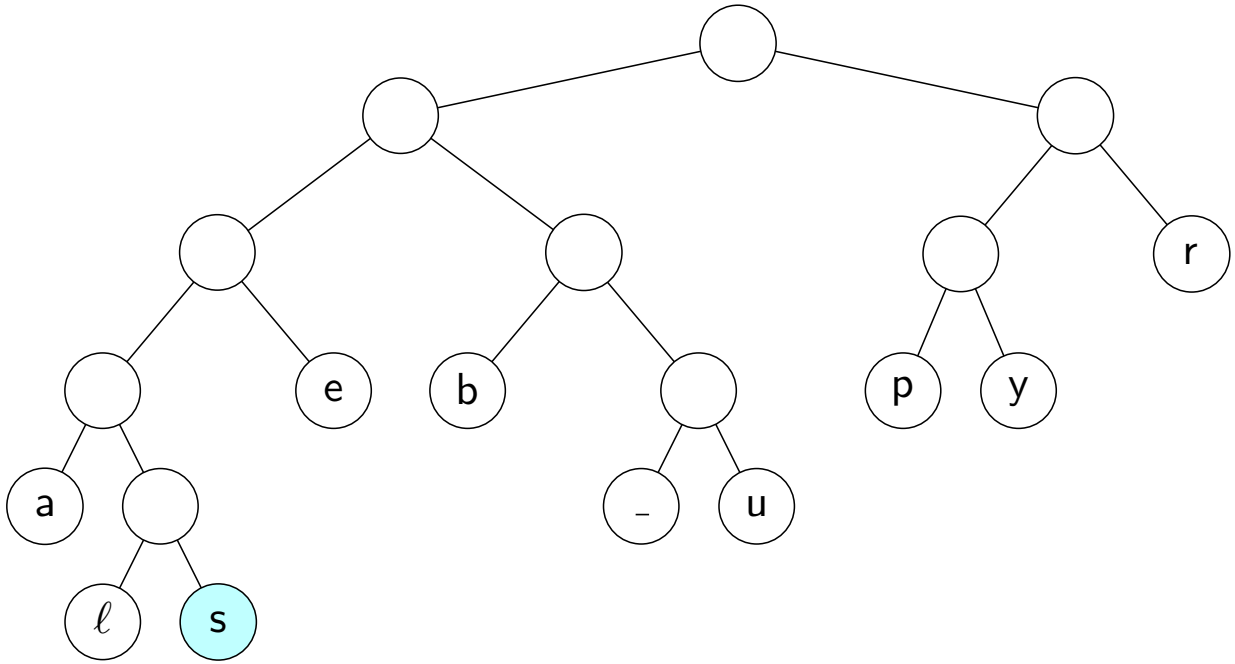
Compressed Data

```
1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 1 0 0 0 1 1 1 1 1 1 0 1 0 1
1 0 0 1 0 0 0 0 1 0 0 1 1 1 0 0 1 0 1 0 0 0 1 1 1 1 1 1 0
1 0 1 1 0 1 0 0 0 0 1 0 0 0 0 1 1
```

Decompressed Data: `ra`

Each step requires constant time, so the entire process is $\Theta(n)$ on n bits of compressed data.

Decoding Prefix-Free Codes (17)



Compressed Data

```
1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 1 0 0 0 1 1 1 1 1 1 0 1 0 1
1 0 0 1 0 0 0 0 1 0 0 1 1 1 0 0 1 0 1 0 0 0 1 1 1 1 1 1 0
1 0 1 1 0 1 0 0 0 0 1 0 0 0 0 1 1
```

Decompressed Data: `ras`

Each step requires constant time, so the entire process is $\Theta(n)$ on n bits of compressed data.

Decoding Prefix-Free Codes (18)

```
 1: procedure DECODE(root, data_source)
 2:     node ← root
 3:     while data_source has compressed bits remaining do
 4:         b ← Next bit from data_source
 5:         if b = 0 then
 6:             node ← node.left
 7:         else
 8:             node ← node.right
 9:         end if
10:         if node is a leaf then
11:             Output the symbol stored in node
12:             node ← root
13:         end if
14:     end while
15: end procedure
```

The pseudocode above gives a sketch of the actual decoding process (but does not include the process of building the tree from the symbol table).

Decoding Prefix-Free Codes (19)

```
 1: procedure DECODE(root, data_source)
 2:     node ← root
 3:     while data_source has compressed bits remaining do
 4:         b ← Next bit from data_source
 5:         if b = 0 then
 6:             node ← node.left
 7:         else
 8:             node ← node.right
 9:         end if
10:         if node is a leaf then
11:             Output the symbol stored in node
12:             node ← root
13:         end if
14:     end while
15: end procedure
```

The formal running time is $\Theta(n + m)$ where n is the number of compressed bits and m is the total number of decompressed bits (or characters). With single-character symbols, this reduces to $\Theta(n)$.

Decoding Prefix-Free Codes (20)

Compressed Data

```
1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 1 0 0 0 1 1 1 1 1 1 0 1 0 1
1 0 0 1 0 0 0 0 1 0 0 1 1 1 0 0 1 0 1 0 0 0 1 1 1 1 1 1 0
1 0 1 1 0 1 0 0 0 0 1 0 0 0 0 1 1
```

Symbol Table

_	0110	a	0000	b	010	e	001	l	00010
p	100	r	11	s	00011	u	0111	y	101

Question: What is the complexity of building the encoding tree from the symbol table?

Decoding Prefix-Free Codes (21)

Compressed Data

```
1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 1 0 0 0 1 1 1 1 1 1 0 1 0 1
1 0 0 1 0 0 0 0 1 0 0 1 1 1 0 0 1 0 1 0 0 0 1 1 1 1 1 1 0
1 0 1 1 0 1 0 0 0 0 1 0 0 0 0 1 1
```

Symbol Table

_	0110	a	0000	b	010	e	001	l	00010
p	100	r	11	s	00011	u	0111	y	101

As shown earlier, we can build the tree by iterating through each symbol and building the path given by the bit sequence. The total running time is then the total number of bits across all encodings (36 in the example above).

Decoding Prefix-Free Codes (22)

Compressed Data

```
1 1 0 0 0 0 0 0 0 1 1 1 0 0 0 1 0 0 0 1 1 1 1 1 1 0 1 0 1
1 0 0 1 0 0 0 0 1 0 0 1 1 1 0 0 1 0 1 0 0 0 1 1 1 1 1 1 0
1 0 1 1 0 1 0 0 0 0 1 0 0 0 0 1 1
```

Symbol Table

_	0110	a	0000	b	010	e	001	l	00010
p	100	r	11	s	00011	u	0111	y	101

A looser bound is $O(sk)$ where s is the number of symbols and k is the maximum length of any symbol's encoding.

Sources

- Slides by B. Bird, 2019.
- The photograph on Slides 10 - 13 was originally taken from Wikimedia Commons (`https://commons.wikimedia.org/wiki/File:Pear_in_tree_0465.jpg`). The image is licensed under a Creative Commons license which allows derivative works (full details are available at the link above).
- The text of "A Tale of Two Cities" is in the public domain and can be obtained from a variety of sources. The version used here was obtained from Project Gutenberg (`http://www.gutenberg.org`).