Indexing/Compression

Last Time

- Text processing:
  - Parsing – idenNfying important parts of document.
  - Tokenizing – separaNng text into words.
  - Stopping – removing very common words.
  - Stemming – reducing words to common stems.

- Purpose: get text documents into a state ready for indexing.

Indexes

- *Indexes* are data structures designed to make search faster
- Text search has unique requirements, which leads to unique data structures
- Most common data structure is *inverted index*
  - general name for a class of structures
  - "inverted" because documents are associated with words, rather than words with documents
    - similar to a *concordance*

Indexes and Ranking

- Indexes are designed to support *search*
  - faster response time, supports updates
- Text search engines use a particular form of search: *ranking*
  - documents are retrieved in sorted order according to a score computed using the document representation, the query, and a *ranking algorithm*
- What is a reasonable abstract model for ranking?
  - enables discussion of indexes without details of retrieval model

Matching Documents and Queries

- What are the important properties of a document that indicate that it might be relevant to a query?
  - Query terms are in the document
  - Terms related to query terms are in the document
  - Contains query terms that are rare in collection
  - Lots of pages that contain query terms link to it
  - Links to other pages that contain query terms
  - Query terms close together in document
  - Date of last document update
  - Popularity of document

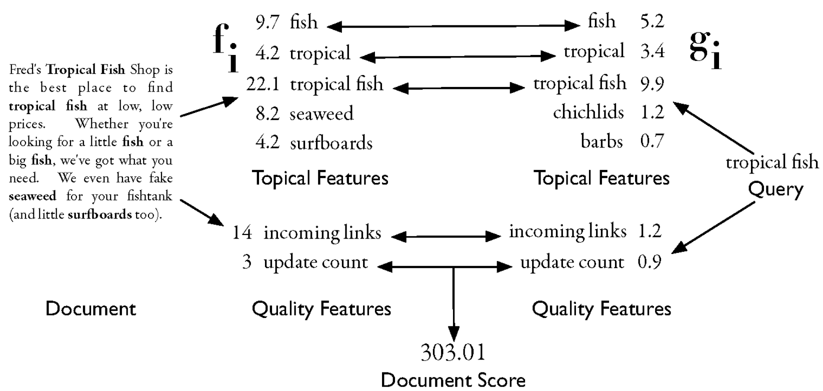
Scoring Documents

- Given those features, how would you score a document for a query?
  - Does it need to contain all the terms? Some?
  - Which features are more important?
  - How do you express their importance?
  - Add features? Multiply?

Concrete Model

$$R(Q, D) = \sum_i g_i(Q) f_i(D)$$

f_i is a document feature function
g_i is a query feature function



Storing Information

What information does the index need to contain to make the scoring happen quickly?

- Important features:
  - Terms are in document.
  - Query terms are rare.
  - Query terms are close together.
- Feature importance:
  - Number of times term appears.
  - Length of document.
  - Rarity of query term.
  - Number of words between query terms.
  - Number of times they occur together versus apart.

Number of times each term appears in each document (term frequency).
Number of documents each term appears in (document frequency).
Length of documents.

Example "Collection"

S_1 Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species.

S_2 Fishkeepers often use the term tropical fish to refer only those requiring fresh water, with saltwater tropical fish referred to as marine fish.

S_3 Tropical fish are popular aquarium fish, due to their often bright coloration.

S_4 In freshwater fish, this coloration typically derives from iridescence, while salt water fish are generally pigmented.

Four sentences from the Wikipedia entry for *tropical fish*

Simple Index Model

	tropic	fish	include	found	environ	around	world	both	freshwat	salt	water	speci	...	length
S_1	2	2	2	1	1	1	1	1	1	1	1	1	0	15
S_2	2	3	0	0	0	0	0	0	0	0	1	0	12	18
S_3	1	2	0	0	0	0	0	0	0	0	0	0	7	10
S_4	0	2	0	0	0	0	0	0	1	1	1	0	8	13
doc. freq.	3	4	1	1	1	1	1	1	2	2	3	1	...	56

- Stores term frequencies in documents, document frequencies of terms, document lengths.
- What is the problem?
  - Suppose integers require 4 bytes of storage.
  - As collection size increases, space requirements grow rapidly.
    - 1M docs with 2M terms = 7451 Gb of disk space!
  - Do we really need to store all those 0s?

Inverted Index

- Each index term is associated with an *inverted list*
  - Contains lists of documents, or lists of word occurrences in documents, and other information
  - Each entry is called a *posting*
  - The part of the posting that refers to a specific document or location is called a *pointer*
  - Each document in the collection is given a unique number
  - Lists are usually *document-ordered* (sorted by document number)

Simple Inverted Index

and	1	only	2
aquarium	3	pigmented	4
are	3 4	popular	3
around	1	refer	2
as	2	referred	2
both	1	requiring	2
bright	3	salt	1 4
coloration	3 4	saltwater	2
derives	4	species	1
due	3	term	2
environments	1	the	1 2
fish	1 2 3 4	their	3
fishkeepers	2	this	4
found	1	those	2
fresh	2	to	2 3
freshwater	1 4	tropical	1 2 3
from	4	typically	4
generally	4	use	2
in	1 4	water	1 2 4
include	1	while	4
including	1	with	2
iridescence	4	world	1
marine	2		
often	2 3		

Inverted Index with counts

- supports better ranking algorithms

and	1:1			
aquarium	3:1			
are	3:1	4:1		
around	1:1			
as	2:1			
both	1:1			
bright	3:1			
coloration	3:1	4:1		
derives	4:1			
due	3:1			
environments	1:1			
fish	1:2	2:3	3:2	4:2
fishkeepers	2:1			
found	1:1			
fresh	2:1			
freshwater	1:1	4:1		
from	4:1			
generally	4:1			
in	1:1	4:1		
include	1:1			
including	1:1			
iridescence	4:1			
marine	2:1			
often	2:1	3:1		

only	2:1		
pigmented	4:1		
popular	3:1		
refer	2:1		
referred	2:1		
requiring	2:1		
salt	1:1	4:1	
saltwater	2:1		
species	1:1		
term	2:1		
the	1:1	2:1	
their	3:1		
this	4:1		
those	2:1		
to	2:2	3:1	
tropical	1:2	2:2	3:1
typically	4:1		
use	2:1		
water	1:1	2:1	4:1
while	4:1		
with	2:1		
world	1:1		

Inverted Index with positions

- supports proximity matches

and	1,15								
aquarium	3,5								
are	3,3	4,14							
around	1,9								
as	2,21								
both	1,13								
bright	3,11								
coloration	3,12	4,5							
derives	4,7								
due	3,7								
environments	1,8								
fish	1,2	1,4	2,7	2,18	2,23	3,2	3,6	4,3	4,13
fishkeepers	2,1								
found	1,5								
fresh	2,13								
freshwater	1,14	4,2							
from	4,8								
generally	4,15								
in	1,6	4,1							
include	1,3								
including	1,12								
iridescence	4,9								

marine	2,22				
often	2,2	3,10			
only	2,10				
pigmented	4,16				
popular	3,4				
refer	2,9				
referred	2,19				
requiring	2,12				
salt	1,16	4,11			
saltwater	2,16				
species	1,18				
term	2,5				
the	1,10	2,4			
their	3,9				
this	4,4				
those	2,11				
to	2,8	2,20	3,8		
tropical	1,1	1,7	2,6	2,17	3,1
typically	4,6				
use	2,3				
water	1,17	2,14	4,12		
while	4,10				
with	2,15				
world	1,11				

Proximity Matches

- Matching phrases or words within a window
  - e.g., "`tropical fish`", or “find tropical within 5 words of fish”
- Word positions in inverted lists make these types of query features efficient
  - e.g.,

tropical	1,1		1,7	2,6	2,17		3,1			
fish	1,2	1,4		2,7	2,18	2,23	3,2	3,6	4,3	4,13

Index Construction

- Simple in-memory indexer

```
procedure BuildIndex(D)                    ▷ D is a set of text documents
    I ← HashTable()                        ▷ Inverted list storage
    n ← 0                                  ▷ Document numbering
    for all documents d ∈ D do
        n ← n + 1
        T ← Parse(d)                       ▷ Parse document into tokens
        Remove duplicates from T
        for all tokens t ∈ T do
            if d ∉ I then
                I_d ← Array()
            end if
            I_d.append(n)
        end for
    end for
    return I
end procedure
```

Simple Indexing Example

tropical	1 2
fish	1 2
include	1
found	1
environments	1
around	1
world	1
both	1
freshwater	1
salt	1
water	1
species	1
fishkeepers	2
often	2
use	2
term	2

S_1 Tropical fish include fish found in tropical environments around the world, including both freshwater and salt water species.

S_2 Fishkeepers often use the term tropical fish to refer only those requiring fresh water, with saltwater tropical fish referred to as marine fish.

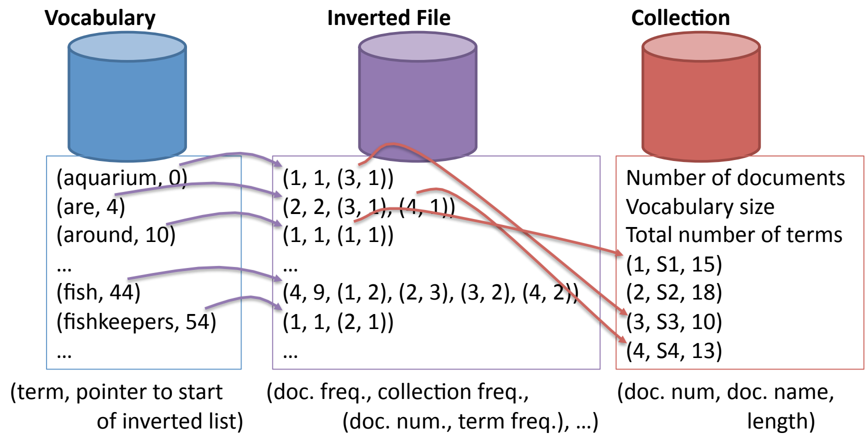
S_3 Tropical fish are popular aquarium fish, due to their often bright coloration.

S_4 In freshwater fish, this coloration typically derives from iridescence, while salt water fish are generally pigmented.

Auxiliary Structures

- Inverted lists usually stored together in a single file for efficiency
  - *Inverted file*
- *Vocabulary* or *lexicon*
  - Contains a lookup table from index terms to the byte offset of the inverted list in the inverted file
  - Either hash table in memory or B-tree for larger vocabularies
- Term statistics stored at start of inverted lists
- Collection statistics stored in separate file

Inverted File + Auxiliary Structures



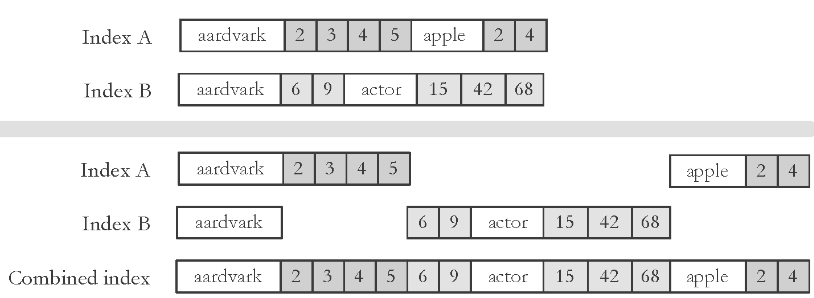
Size of Inverted Lists

- Inverted lists are smaller than our matrix idea, but still can be quite large.
  – 1M documents with 2M terms = 3.7Gb space.
  – Heaps' law and Zipf's law can be used to predict the space required by an inverted index.
- Too big to hold in memory.
- Two solutions:
  – Merging on disk.
  – Compression.

Merging

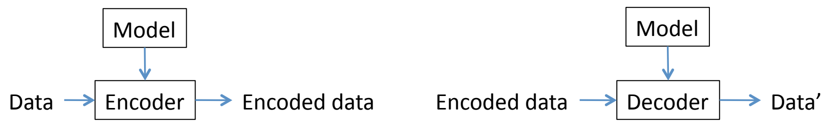
- Merging addresses limited memory problem
  - Build the inverted list structure until memory runs out
  - Then write the partial index to disk, start making a new one
  - At the end of this process, the disk is filled with many partial indexes, which are merged
- Partial lists must be designed so they can be merged in small pieces
  - e.g., storing in alphabetical order

Merging



What is Compression?

- Compression is a type of *encoding* of data.



- The goal is to make the data smaller.
- A very big topic in CS and engineering.
  – We have a full course on data compression.

Types of Compression

- Lossless compression:
  – The encoding preserves all information about the original data.
  – The original data can be recovered completely.
- Lossy compression:
  – The encoding loses some information about the original data.
  – The original data can be recovered approximately.
- Signature file indexes are a type of lossy compression.

Compression in IR

- Text compression:
  - Used to compress vocabulary, document names, original document text.
  - Based on assumptions about language.
- Data compression:
  - Used to compress inverted lists.
  - Not generally based on assumptions, but on observations about the data.

Preliminaries

- "Text" means based on characters.
- What is a character? (Think C, C++)
  - A data type.
  - Generally stores 1 byte.
  - 1 byte = 8 bits.
  - Since each bit can be 0 or 1, one byte can store 2^8 = 256 possible characters.

ASCII Encoding

- ASCII is a common character encoding.
- Each character is represented with 8 bits.
  - A = ASCII 65 = 01000001
  - ¿ = ASCII 168 = 10101000
  - 256 possible characters.
- Decoding: table maps bytes to characters.
- Fish: 01000110 01101001 01110011 01101000
  - 32 bits = 4 bytes.

Fixed Length Codes

- Short bytes: use the smallest number of bits needed to represent all characters.
  - English has 26 letters. How many bits needed?
  - 5 bits can represent 2^5 = 32 letters.
  - 26 letters * 2 cases = 52 characters.
    - Requires 6 bits… or does it?
- Use numbers 1-30 (00001 – 11110) to represent two sets of characters.
  - Use 0 (00000) to toggle the first set (e.g. capital letters).
  - Use 31 (11111) to toggle the second set (e.g. small letters).
- Fish: $\underbrace{00110}_{\text{F}}\ \underbrace{11111}_{\downarrow}\ \underbrace{01001}_{\text{i}}\ \underbrace{10011}_{\text{s}}\ \underbrace{01000}_{\text{h}}$
  - 25 bits, slightly over 3 bytes.

Fixed Length Codes

- Bigram codes: use 8 bits to encode either 1 or 2 characters.
  - *is* would be encoded in 8 bits.
- Use values 0-87 for space, 26 lower case, 26 upper case, 10 numbers, and 25 other characters.
- Use values 88-255 for character pairs.
  - Master (8): blank, A, E, I, O, N, T, U
  - Combining (21): blank, all other letters except JKQXYZ
  - 88 + 8*21 = 256 possibilities encoded
- Fish: 00100000 10101010 00001000
  F is h
  - 24 bits, 3 bytes.

Fixed Length Codes

- *N*-gram codes: same as bigram, but encode character strings of length less than or equal to *n*.
- Select most common strings for 8-bit encoding in advance.
  - Goal: most commonly occurring *n*-grams require only one byte.
- Fish: 00100000 10111010
  F ish
  - 16 bits, 2 bytes.

Fixed Length Summary

- Fixed length codes are generally simple, easy to use, and effective when assumptions are met.
- Limited alphabet size allowed.
- If data does not meet assumptions, compression will not be good.

Restricted Variable Length Codes

- Idea: different characters can have encodings of different lengths.
- Similar to case-shifting in short byte codes:
  - First bit indicates case.
  - 8 most common characters encoded in 4 bits (0xxx)
  - 128 less common characters encoded in 8 bits (1xxxxxxx)
  - First bit tells you how many bits to read next.
- 8 most common English letters are e, t, a, i, n, o, r, s.
- Fish: $\underbrace{10000110}_{F}\ \underbrace{0011}_{i}\ \underbrace{0110}_{s}\ \underbrace{10000100}_{h}$
  - 24 bits, 3 bytes.

Shannon Game

- The President of the United States is Barack …
  - Only one possible option. We don't even need to send the last word to transmit the information.
- The best web search engine is …
  - Many options, but one has high probability. Two others have lower but non-negligible probability. Many others have low probability.
  - We could guess the next word, but we could be wrong.
- Mary was …
  - Happy? angry? tall? Who knows…

Information Content

- The *information content* of a message is a function of how predictable it is.
  - … Obama – very predictable → very low information content if you read U.S. news at all.
  - … Google – somewhat predictable → low (but non-zero) information content.
  - … Queen of England from 1553 to 1558 – unpredictable → high information content: you weren't expecting it.

Encoding Information

- Let p_i be the probability of message *i*.
  - For first example, $p_{Obama} = 1$.
  - For second, suppose $p_{Google} = 0.5$, $p_{Yahoo} = 0.3$, $p_{Microsoft} = 0.15$, $p_{Other} = 0.05$.
  - For third, many possibilities with low probability.
- The number of bits needed to encode *i* is $-\log_2 p_i$.
  - Obama: $-\log_2 1 = 0$ bits.
  - Google: $-\log_2 0.5 = 1$ bit; Yahoo: $-\log_2 0.3 = 1.74$ bits; Microsoft: $-\log_2 0.15 = 2.74$ bits; other = $-\log_2 0.05 = 4.32$ bits.
    - "not Google": $-\log_2 (1 - 0.5) = 1$ bit.

Information Entropy

- The *entropy* of a message is the expected number of bits needed to encode it.
  - Expectation = sum over all possibilities, probability of possibility times value of possibility.
  - Entropy = $H(p) = -\sum_{i=1}^{n} p_i \log_2 p_i$
- First example: H = $-1*\log_2 1 = 0$.
- Second example: H = $-0.5*\log_2 0.5 - 0.3*\log_2 0.3 - 0.15*\log_2 0.15 - 0.05*\log_2 0.05 = 1.65$ bits.
  - Google vs. non-Google: H = -.5*log .5 - .5*log .5 = 1 bit.

Information Theory and Codes

- We have implicitly been using information theory to determine minimum code lengths.
  - Recall short byte codes: characters represented with 5 bits.
  - For alphabet size 26, each letter probability 1/26:
    - $-\log_2 1/26 = 4.7$ bits, so 5 bits necessary.
- Information theory allows us to find more compact representations.
  - Using frequencies of letter occurrences, we can reduce entropy to 3.56 bits or less.
  - Humans can guess the next letter in a sequence accurately; only need 1.3 bits.

Huffman Encoding

- An information-theoretic variable-length code.
- Basic idea: create a tree
  - Calculate the probability of each symbol.
  - Make the two lowest-probability symbols or nodes inherit from a parent node.
    - P(parent) = P(child1) + P(child2)
  - Label lower-probability node 0, other node 1.
  - Iterate until all nodes connected in a tree.
- Path from root to leaf determines code of leaf.

Example

- Say we want to encode a text with the characters a, b,…, g occurring with the following frequencies:

	a	b	c	d	e	f	g
Frequency	37	18	29	13	30	17	6

Fixed-Length Code

	a	b	c	d	e	f	g
Frequency	37	18	29	13	30	17	6
Fixed-length code	000	001	010	011	100	101	110

- Total size is:

  (37 + 18 + 29 + 13 + 30 + 17 + 6) x 3= 450 bits

Variable-Length Code

	a	b	c	d	e	f	g
Frequency	37	18	29	13	30	17	6
Variable-length code	10	011	111	1101	00	010	1100

- Total size is:

  $37x2 + 18x3 + 29x3 + 13x4 + 30x2 + 17x3 + 6x4 = 402$ bits

- A savings of approximately 11%

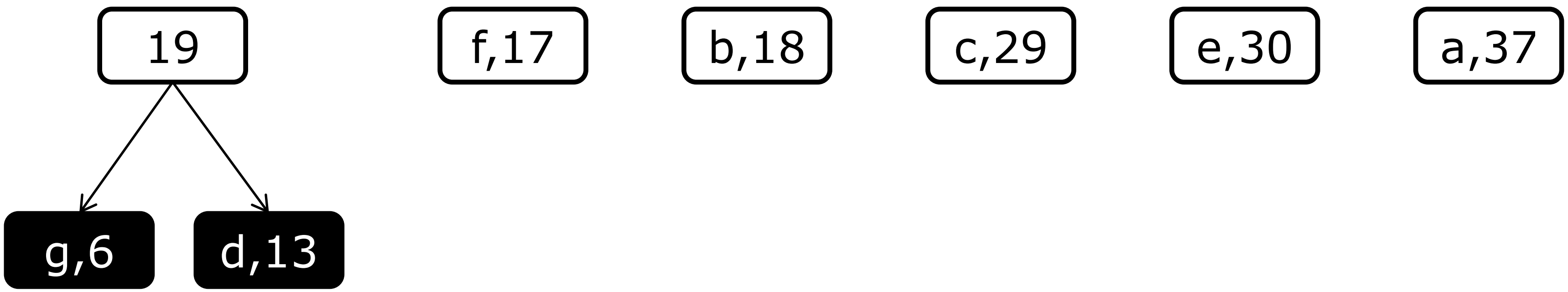
Constructing a Huffman Code

g,6 d,13 f,17 b,18 c,29 e,30 a,37

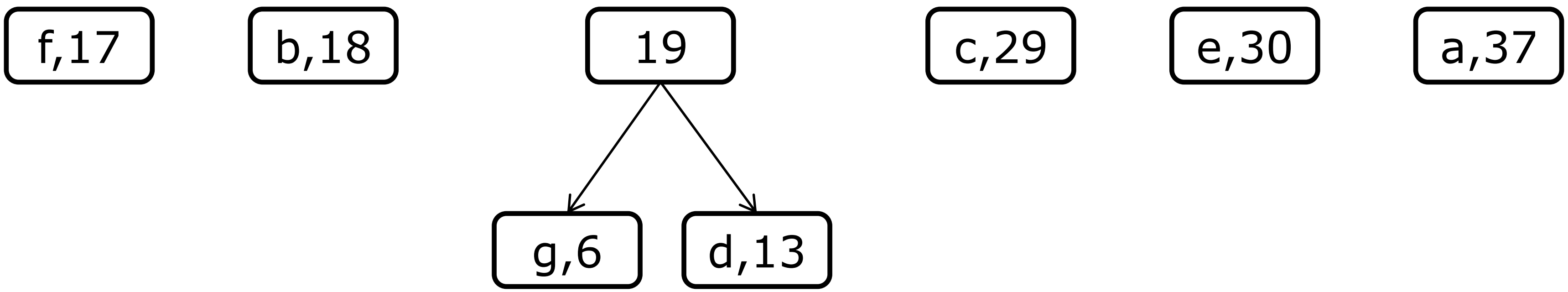
Constructing a Huffman Code

g,6 d,13 f,17 b,18 c,29 e,30 a,37

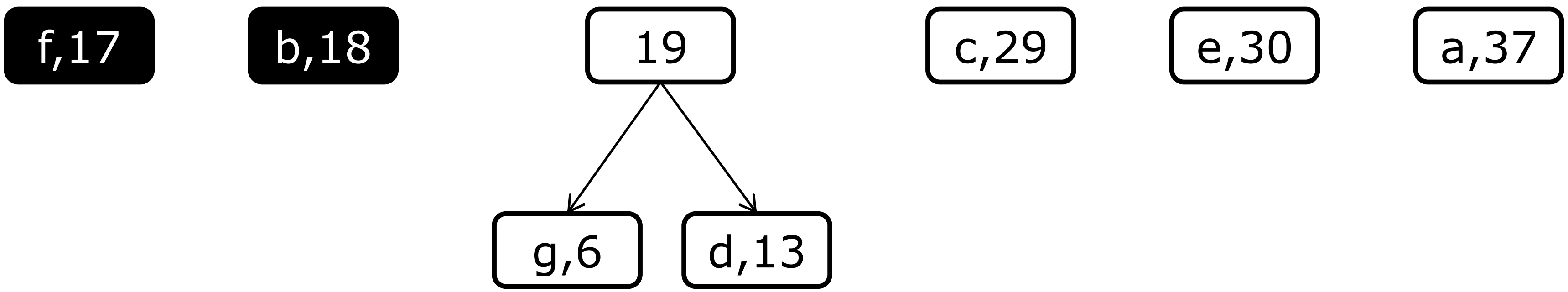
Constructing a Huffman Code



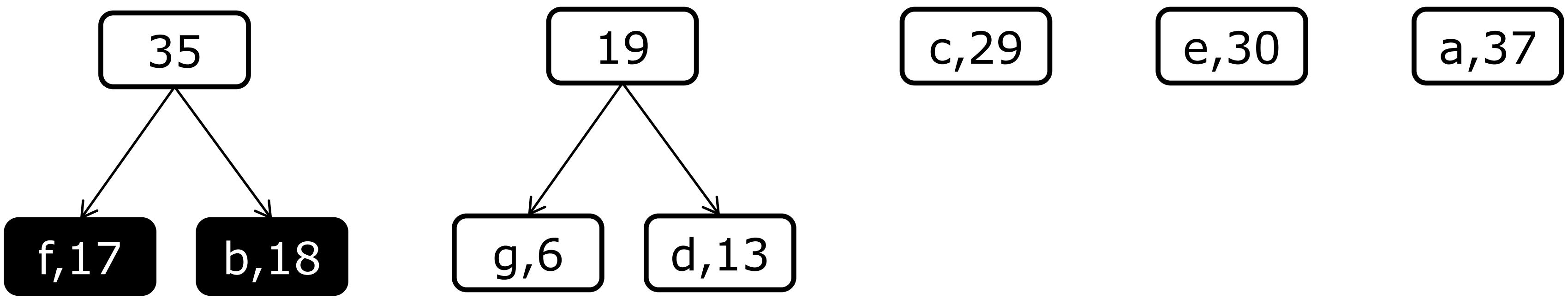
Constructing a Huffman Code



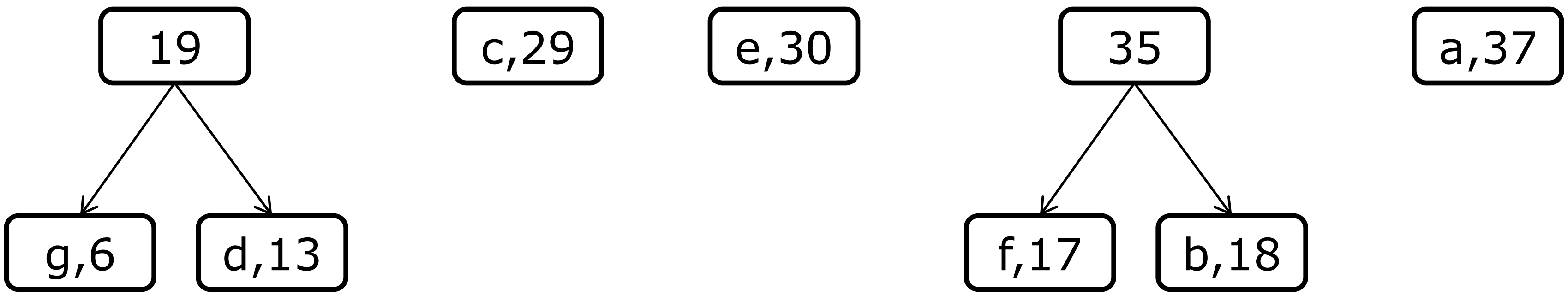
Constructing a Huffman Code



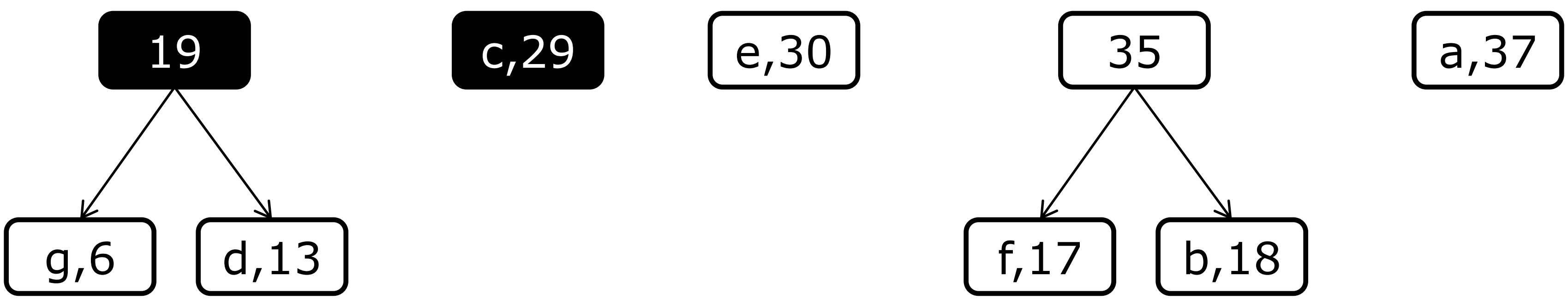
Constructing a Huffman Code



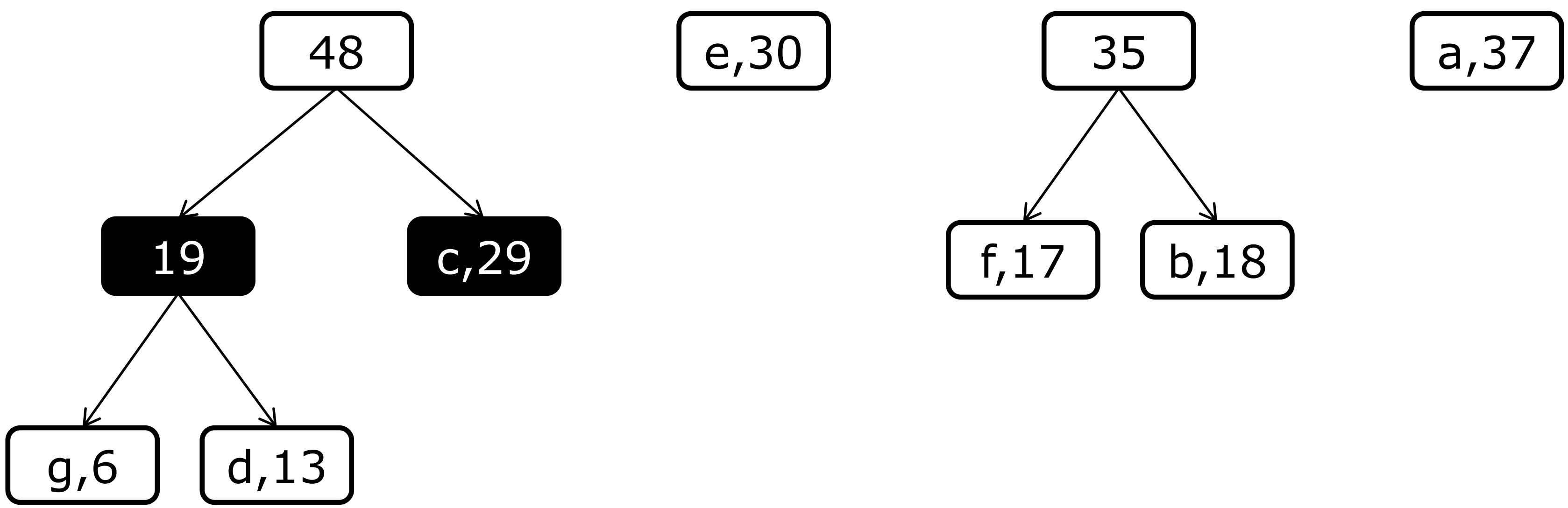
Constructing a Huffman Code



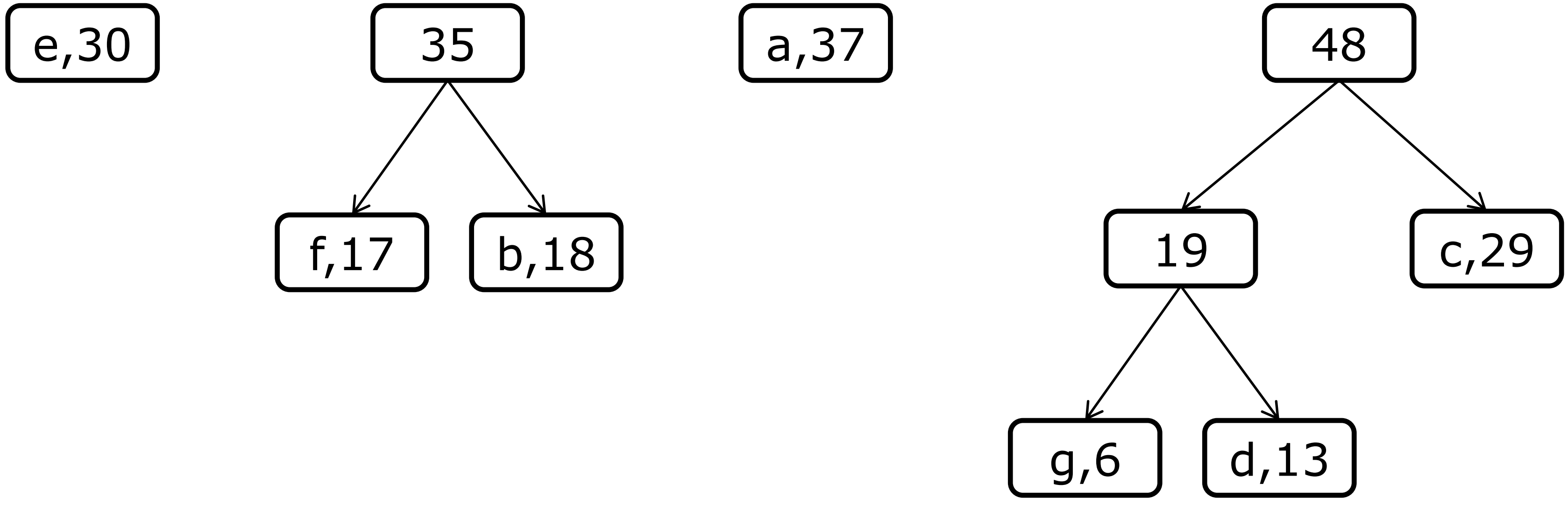
Constructing a Huffman Code



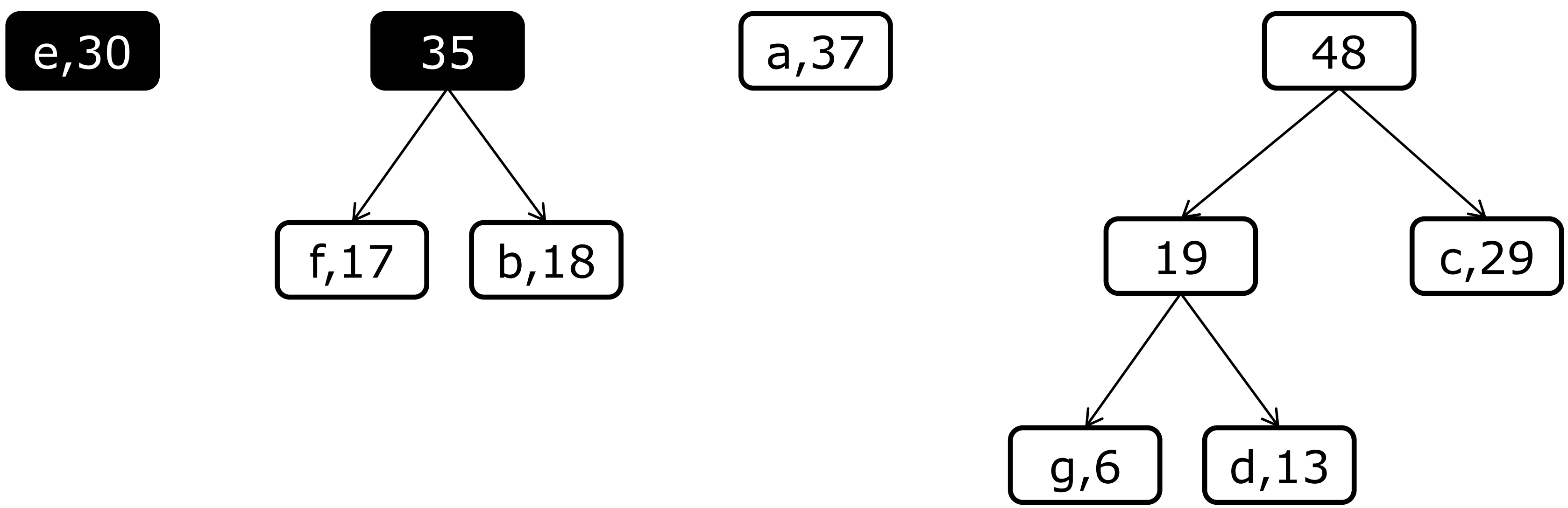
Constructing a Huffman Code



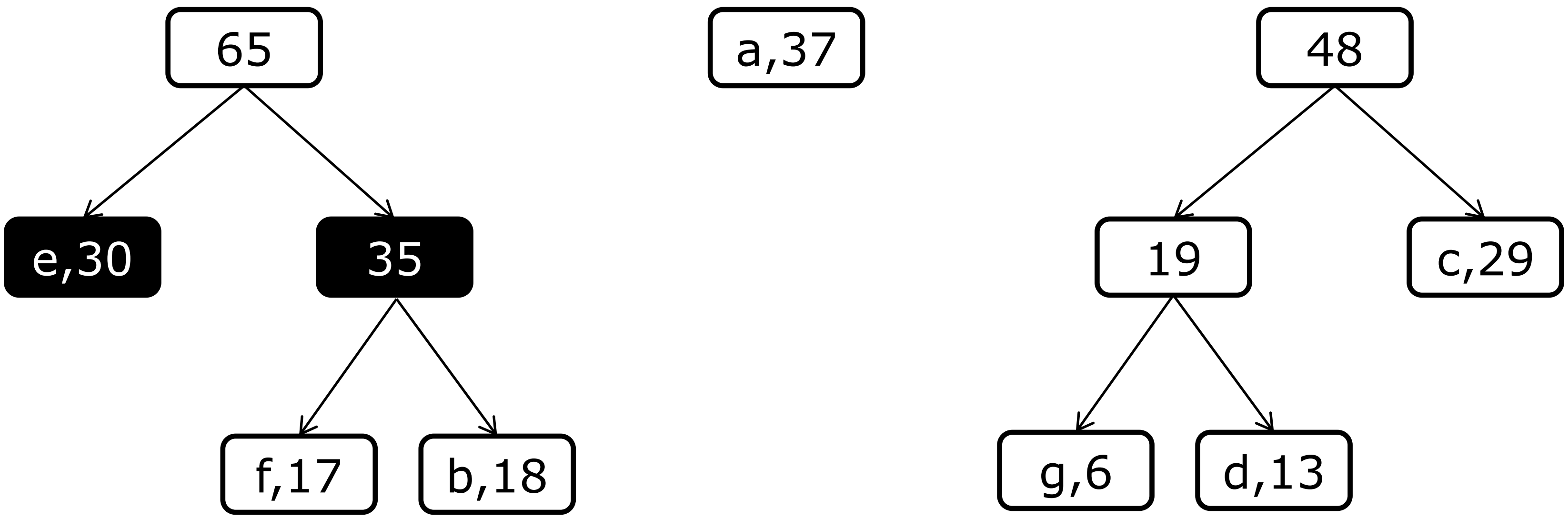
Constructing a Huffman Code



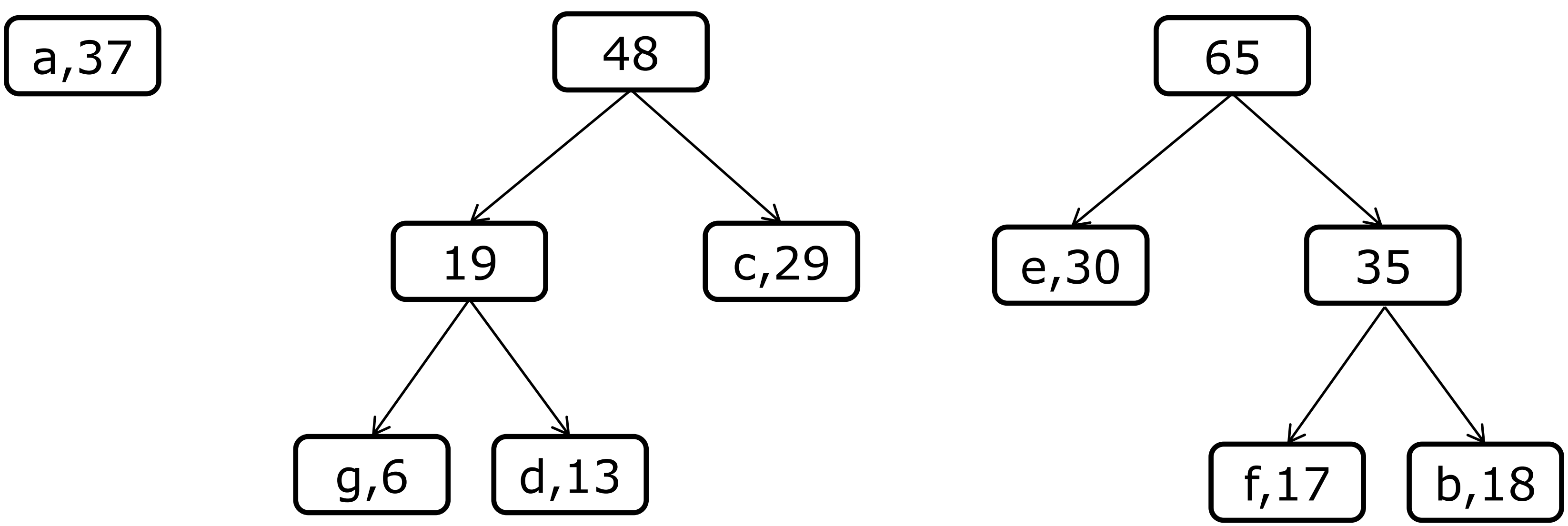
Constructing a Huffman Code



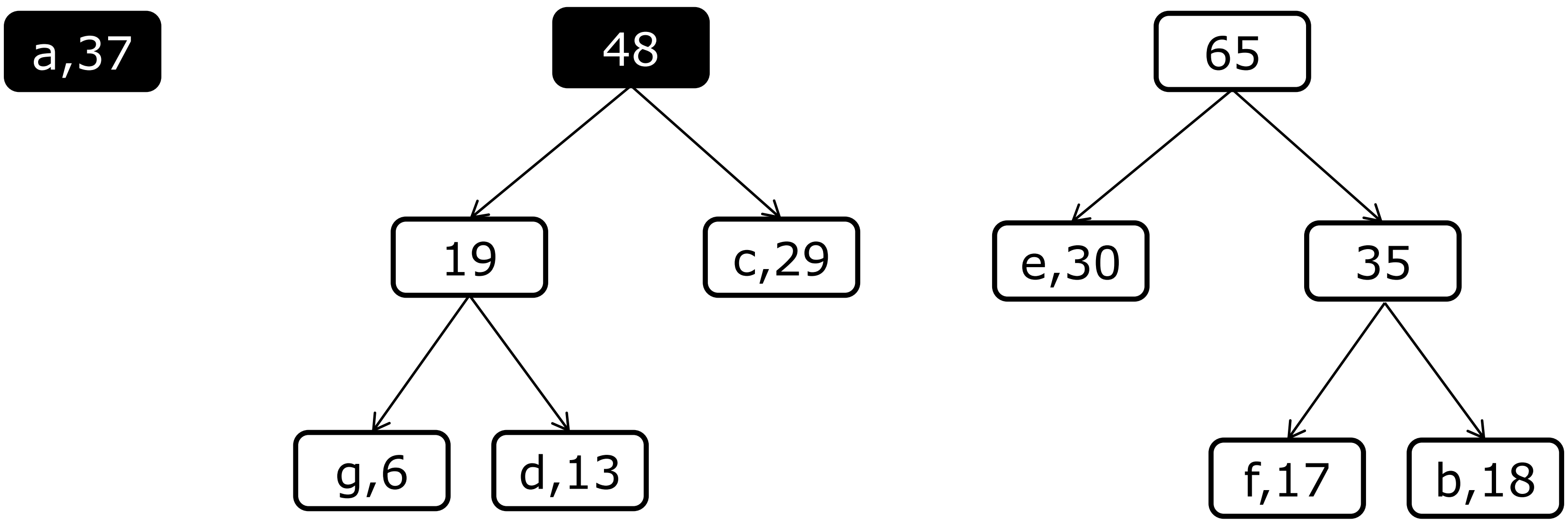
Constructing a Huffman Code



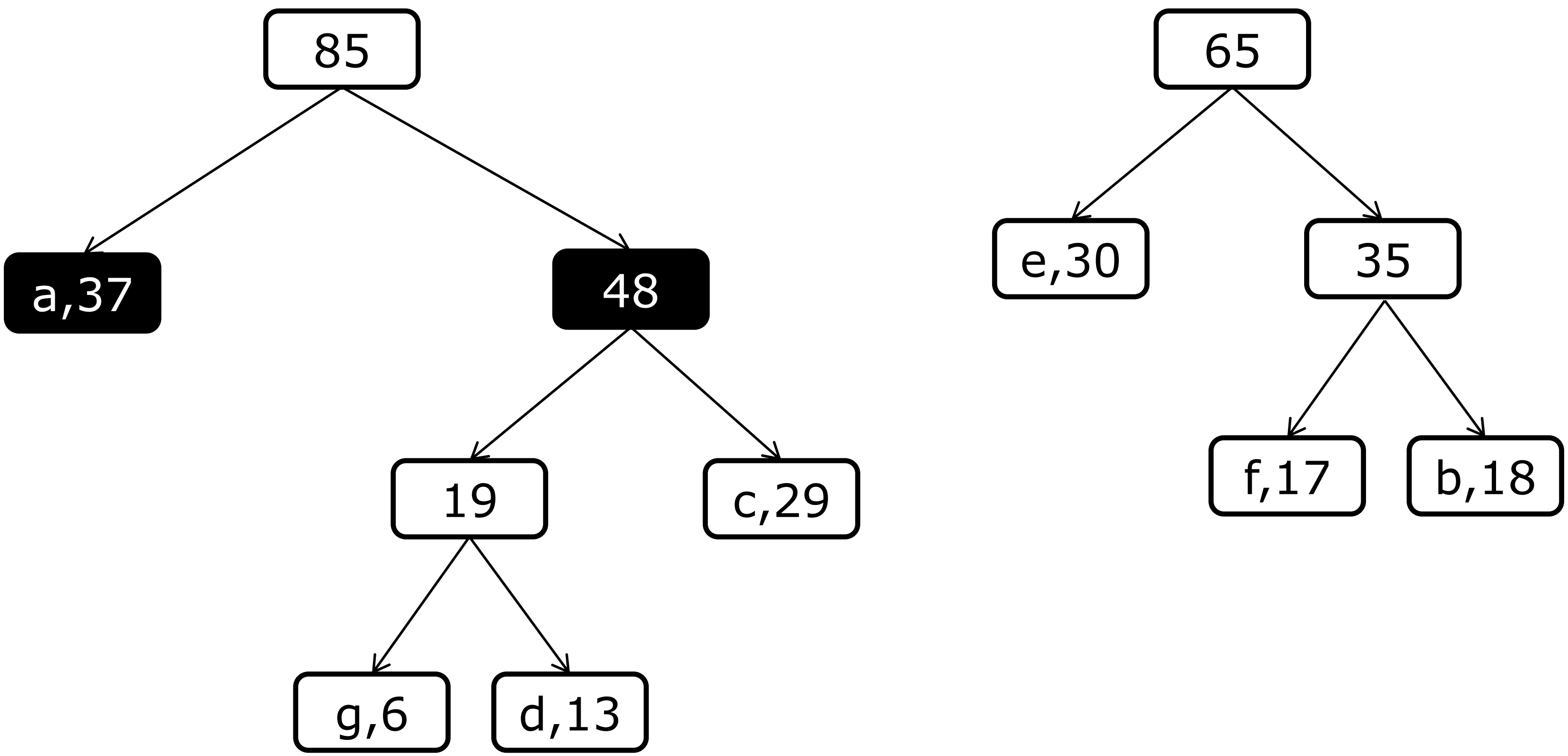
Constructing a Huffman Code



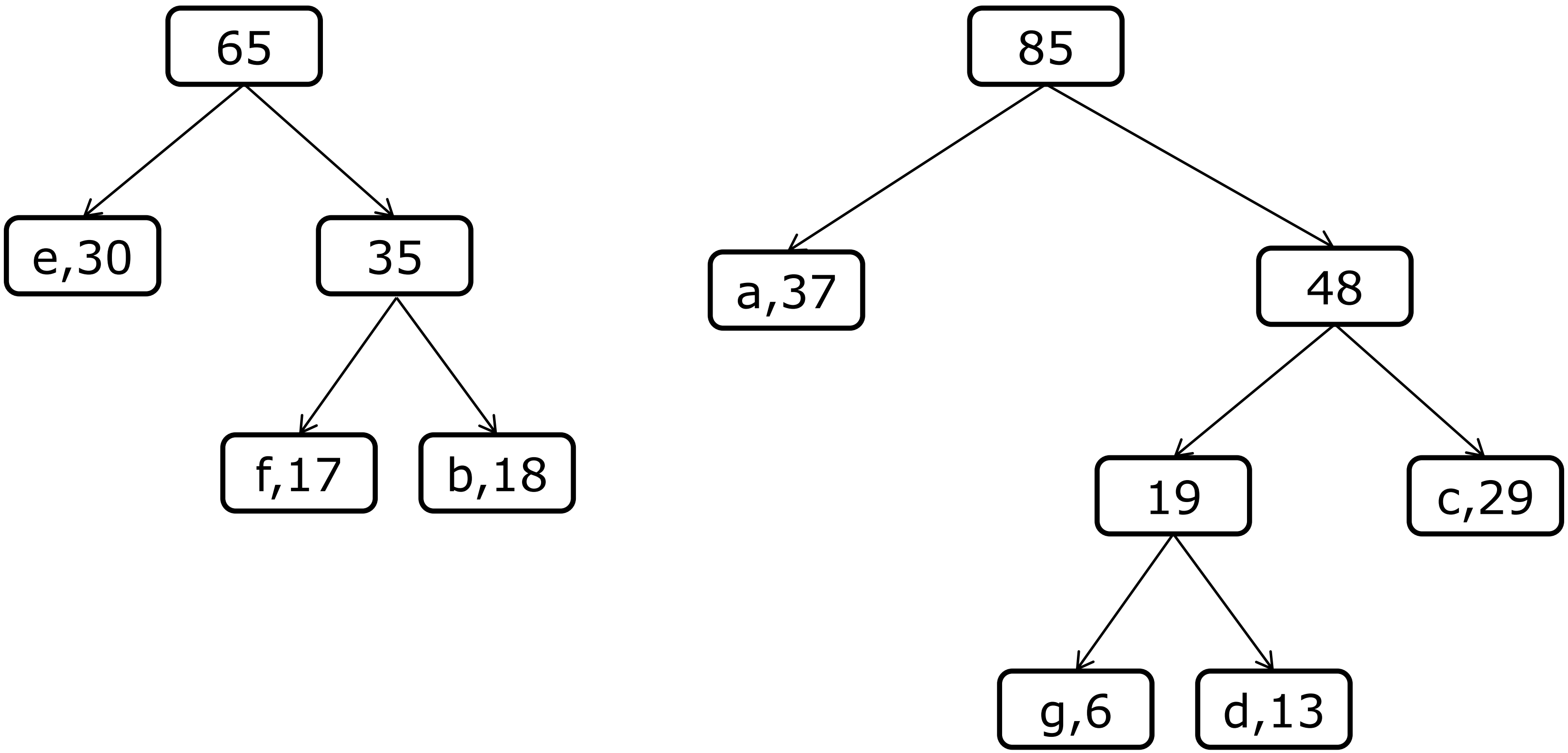
Constructing a Huffman Code



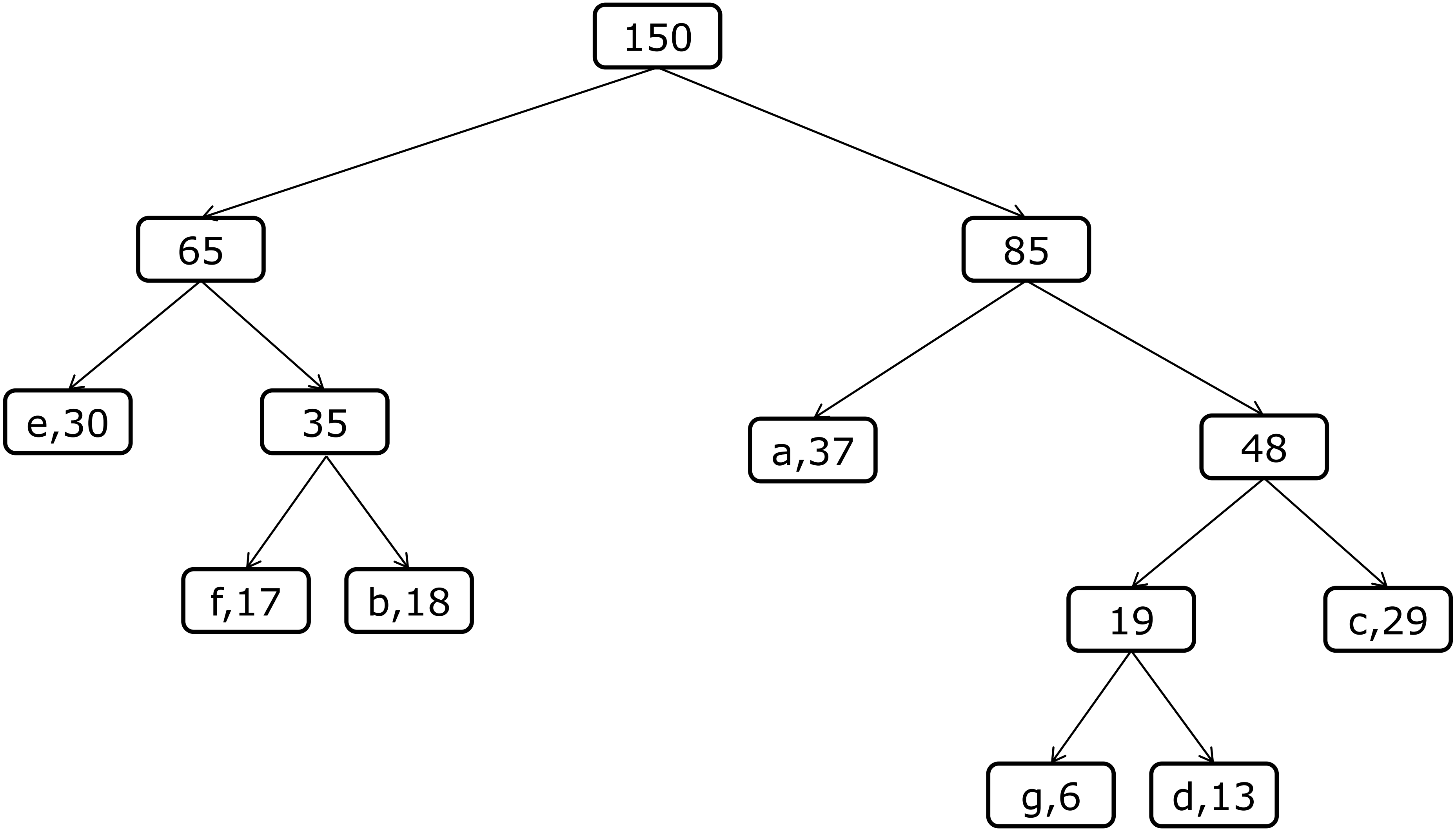
Constructing a Huffman Code



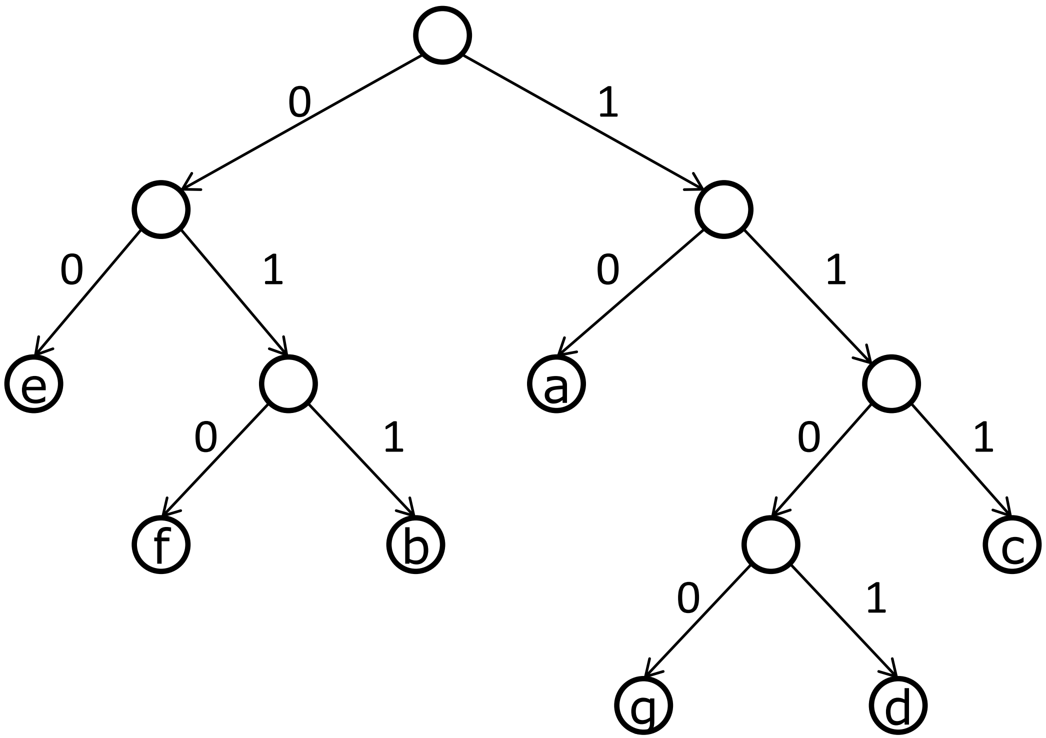
Constructing a Huffman Code



Constructing a Huffman Code



Resulting Code



a	10
b	011
c	111
d	1101
e	00
f	010
g	1100

Huffman Codes

- Huffman codes are “prefix free”: no code is a prefix of another.
  - Uniquely decodable; lossless compression.
- They come very close to the limits of compressibility proved by Shannon.
- Decoding somewhat inefficient.
  - Must store entire tree in memory; process encoded data bit by bit.

Lempel-Ziv Compression

- A dictionary-based approach to variable length coding.
- Build a dictionary as text is encountered in the file.
  - If Zipf's law is obeyed, the dictionary will be good.
- Dictionary does not need to be stored, as both encoder and decoder know how to create it.
- Used in many modern compression programs:
  - gzip, Unix compress, zip.
  - And some compressed file formats like PNG.

Lempel-Ziv

example : and

ASCII: 0 1 1 0 0 0 0 1 0 1 1 0 1 1 1 0 0 1 1 0 0 1 0 0

Lempel-Ziv

example : and

ASCII: 0 |1 |1 0| 0 0 |0 1| 0 1 1| 0 1 1 1| 0 0 1| 1 0 0 |1 0 0

1. Parse the string from left to right into unique substrings

Lempel-Ziv

example : and

ASCII: 0 |1 |1 0| 0 0 |0 1| 0 1 1| 0 1 1 1| 0 0 1| 1 0 0 |1 0 0

bin 1 2 3 4 5 6 7 8 9 10

EN (-,0) (0,1) (2,0) (1,0) (1,1) (5,1) (6,1) (4,1) (3,0) (9,?)

1. Parse the string from left to right into unique substrings
2. Encode each substring as

   position of prefix seen in past + suffix bit

Lempel-Ziv

example : and

ASCII:	0	1	1 0	0 0	0 1	0 1 1	0 1 1 1	0 0 1	1 0 0	1 0 0
bin	1	2	3	4	5	6	7	8	9	10
EN	(-,0)	(0,1)	(2,0)	(1,0)	(1,1)	(5,1)	(6,1)	(4,1)	(3,0)	(9,?)

Number of bits:

- Prefix location = $\lceil \log_2(\#\text{ of Bins}) \rceil$, e.g. $\lceil \log_2 10 \rceil = 4\,bits$
- Suffix bit = 1 bit

Improved Lempel-Ziv

example : and

ASCII: 0 | 1 | 1 0 | 0 0 | 0 1 | 0 1 1 | 0 1 1 1 | 0 0 1 | 1 0 0 | 1 0 0

EN (-,0) (0,1) (1,0) (3,0) (4,1) (1,1) (1,1) (4,1) (6,0) (1,?)

1. Parse the string from left to right into unique substrings
2. Encode each substring as
   relative position of prefix seen in past + suffix bit

Improved Lempel-Ziv

example : and

ASCII: 0 | 1 | 1 0 | 0 0 | 0 1 | 0 1 1 | 0 1 1 1 | 0 0 1 | 1 0 0 | 1 0 0

EN (-,0) (0,1) (1,0) (3,0) (4,1) (1,1) (1,1) (4,1) (6,0) (1,?)

Question: If I am on the 4th substring what is the most I need to go back?

Answer: 3 bins, thus I need $\log_2(3+1)=2$ bits to encode prefix

Improved Lempel-Ziv

example : and

ASCII: 0 | 1 | 1 0 | 0 0 | 0 1 | 0 1 1 | 0 1 1 1 | 0 0 1 | 1 0 0 | 1 0 0

EN (-,0) (0,1) (1,0) (3,0) (4,1) (1,1) (1,1) (4,1) (6,0) (1,?)

Question: If I am on the 8th substring what is the most I need to go back?

Answer: 7 bins, thus I need $\log_2(7+1)=3$ bits to encode prefix

Improved Lempel-Ziv

example : and

ASCII:	0	1	1 0	0 0	0 1	0 1 1	0 1 1 1	0 0 1	1 0 0	1 0 0
bin	1	2	3	4	5	6	7	8	9	10
EN	(-,0)	(0,1)	(1,0)	(3,0)	(4,1)	(1,1)	(1,1)	(4,1)	(6,0)	(1,?)

0 bits

1 bit

2 bits

3 bits

4 bits

Use $\lceil \log_2 n \rceil$ bits to encode the back pointer (relative position of prefix) when you are in substring n

Improved Lempel-Ziv

example : and

ASCII:	0	1	1 0	0 0	0 1	0 1 1	0 1 1 1	0 0 1	1 0 0	1 0 0
bin	1	2	3	4	5	6	7	8	9	10
	0 bits	1 bit		2 bits				3 bits		4 bits
EN	(-,0)	(0,1)	(1,0)	(3,0)	(4,1)	(1,1)	(1,1)	(4,1)	(6,0)	(1,?)
	(-,0)	(0,1)	(01,0)	(11,0)	(100,1)	(001,1)	(001,1)	(100,1)	(0110,0)	(0001,?)

CODE 001010110100100110011100101100001

Improved Lempel-Ziv

- Decode

example : $001010110100100110011100101100001$

0 | 01 | 010 | 110 | 1001 | 0011 | 0011 | 1001 | 01100 | 0001

1. Parse the string from left to right into substrings
   - for the substring n, split after $\lceil \log_2 n \rceil + 1$ bits

Improved Lempel-Ziv

- Decode

example : 0010101101001001100111001011000001

0 | 01 | 010 | 110 | 1001 | 0011 | 0011 | 1001 | 01100 | 0001

(-,0) (0,1) (01,0) (11,0) (100,1) (001,1) (001,1) (100,1) (0110,0) (0001,?)

1. Parse the string from left to right into substrings
2. Split each bin to (prefix code, suffix bit)

Improved Lempel-Ziv

- Decode

example : $001010110100100110011100101100001$

0 | 01 | 010 | 110 | 1001 | 0011 | 0011 | 1001 | 01100 | 0001

(-,0) (0,1) (01,0) (11,0) (100,1) (001,1) (001,1) (100,1) (0110,0) (0001,?)

(-,0) (0,1) (1,0) (3,0) (4,1) (1,1) (1,1) (4,1) (6,0) (1,?)

1. Parse the string from left to right into substrings
2. Split each bin to (prefix code, suffix bit)
3. Decode prefix code into number

Improved Lempel-Ziv

- Decode

example : 00101011010010011001110010110000001

0	01	010	110	1001	0011	0011	1001	01100	0001
(-,0)	(0,1)	(01,0)	(11,0)	(100,1)	(001,1)	(001,1)	(100,1)	(0110,0)	(0001,?)
(-,0)	(0,1)	(1,0)	(3,0)	(4,1)	(1,1)	(1,1)	(4,1)	(6,0)	(1,?)
0	1	10							

1. Parse the string from left to right into substrings
2. Split each bin to (prefix code, suffix bit)
3. Decode prefix code into number
4. Reconstruct the ASCII : prefix + suffix bit

Improved Lempel-Ziv

- Decode

example : 00101011010010011001110010110000 01

0 | 01 | 010 | 110 | 1001 | 0011 | 0011 | 1001 | 01100 | 0001

(-,0) (0,1) (01,0) (11,0) (100,1) (001,1) (001,1) (100,1) (0110,0) (0001,?)

(-,0) (0,1) (1,0) (3,0) (4,1) (1,1) (1,1) (4,1) (6,0) (1,?)

0 1 10 00

1. Parse the string from left to right into substrings
2. Split each bin to (prefix code, suffix bit)
3. Decode prefix code into number
4. Reconstruct the ASCII : prefix + suffix bit

Improved Lempel-Ziv

- Decode

example : 00101011010010011001110010110000001

0	01	010	110	1001	0011	0011	1001	01100	0001
(-,0)	(0,1)	(01,0)	(11,0)	(100,1)	(001,1)	(001,1)	(100,1)	(0110,0)	(0001,?)
(-,0)	(0,1)	(1,0)	(3,0)	(4,1)	(1,1)	(1,1)	(4,1)	(6,0)	(1,?)
0	1	10	00	01					

1. Parse the string from left to right into substrings
2. Split each bin to (prefix code, suffix bit)
3. Decode prefix code into number
4. Reconstruct the ASCII : prefix + suffix bit

Improved Lempel-Ziv

- Decode

example : 00101011010010011001110010110000001

0	01	010	110	1001	0011	0011	1001	01100	0001
(-,0)	(0,1)	(01,0)	(11,0)	(100,1)	(001,1)	(001,1)	(100,1)	(0110,0)	(0001,?)
(-,0)	(0,1)	(1,0)	(3,0)	(4,1)	(1,1)	(1,1)	(4,1)	(6,0)	(1,?)
0	1	10	00	01	011				

1. Parse the string from left to right into substrings
2. Split each bin to (prefix code, suffix bit)
3. Decode prefix code into number
4. Reconstruct the ASCII : prefix + suffix bit

Improved Lempel-Ziv

- Decode

example : 0010101101001001100111001011000001

0 | 01 | 010 | 110 | 1001 | 0011 | 0011 | 1001 | 01100 | 0001

(-,0) (0,1) (01,0) (11,0) (100,1) (001,1) (001,1) (100,1) (0110,0) (0001,?)

(-,0) (0,1) (1,0) (3,0) (4,1) (1,1) (1,1) (4,1) (6,0) (1,?)

0 1 10 00 01 011 0111

1. Parse the string from left to right into substrings
2. Split each bin to (prefix code, suffix bit)
3. Decode prefix code into number
4. Reconstruct the ASCII : prefix + suffix bit

Improved Lempel-Ziv

- Decode

example : 001010110100100110011100101100001

0 | 01 | 010 | 110 | 1001 | 0011 | 0011 | 1001 | 01100 | 0001

(-,0)	(0,1)	(01,0)	(11,0)	(100,1)	(001,1)	(001,1)	(100,1)	(0110,0)	(0001,?)
(-,0)	(0,1)	(1,0)	(3,0)	(4,1)	(1,1)	(1,1)	(4,1)	(6,0)	(1,?)
0	1	10	00	01	011	0111	001	100	100

1. Parse the string from left to right into substrings
2. Split each bin to (prefix code, suffix bit)
3. Decode prefix code into number
4. Reconstruct the ASCII : prefix + suffix bit

Improved Lempel-Ziv

- Decode

example : 00101011010010011001110010110000001

0	01	010	110	1001	0011	0011	1001	01100	0001
(-,0)	(0,1)	(01,0)	(11,0)	(100,1)	(001,1)	(001,1)	(100,1)	(0110,0)	(0001,?)
(-,0)	(0,1)	(1,0)	(3,0)	(4,1)	(1,1)	(1,1)	(4,1)	(6,0)	(1,?)
0	1	10	00	01	011	0111	001	100	100

1. Parse the string from left to right into substrings
2. Split each bin to (prefix code, suffix bit)
3. Decode prefix code into number
4. Reconstruct the ASCII : prefix + suffix bit
5. From ASCII to characters