Numeric Representation III

CSCI 2050U - Computer Architecture

Randy J. Fortier
@randy_fortier

OntarioTech UNIVERSITY

Outline

- Errors detection and correction
- Encoding
  - Prefix codes
  - Huffman's algorithm

Detecting and Correcting Errors

CSCI 2050U - Computer Architecture

OntarioTech UNIVERSITY

Error Detection

- The simplest way to detect an error is to use a parity bit
  - A redundant bit to ensure that no bits were changed (0 → 1, 1 → 0)
  - Even parity - set the parity bit such that the total number of 1s is even
  - Odd parity - set the parity bit such that the total number of 1s is odd
  - Can only detect single-bit errors
  - Cannot correct the error

Error Detection

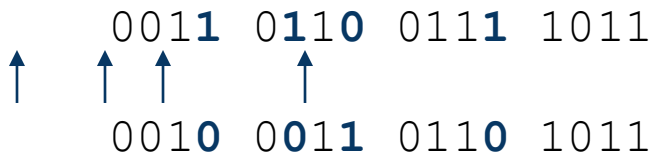
- Example (even parity):
  - Sent: `0000 0110 0`
  - Received: `0000 01`**`0`**`0 0`
  - The total number of 1s is 1, which is odd
  - Therefore, there must have been an error

Error Detection

- Another example (even parity):
  - Sent: `0000 0110 0`
  - Received: `0010 0100 0`
  - The total number of 1s is 2, which is even
  - No error is detected

Hamming Distance

- Hamming distance is the number of symbols in one string that are different from the symbols at the same location in another string
- Example:



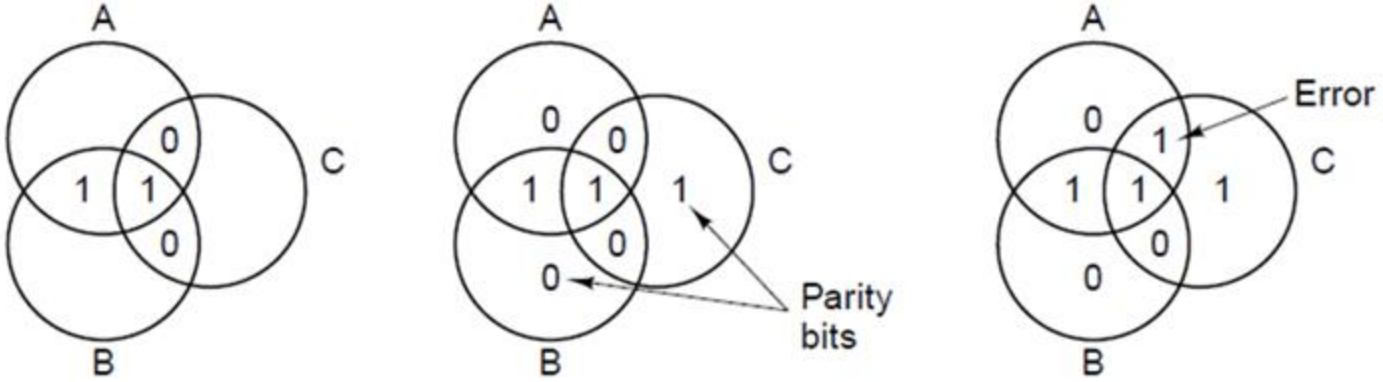
Hamming distance: 4

Hamming Distance

- A parity bit can only detect errors where the hamming distance is 1
- One way to improve our resistance to errors is to choose an encoding where the hamming distance between values is larger
- Example:
  - `000` represents `0`
  - `111` represents `1`
- Now, if we receive `010`, we could assume that this was a `0` value

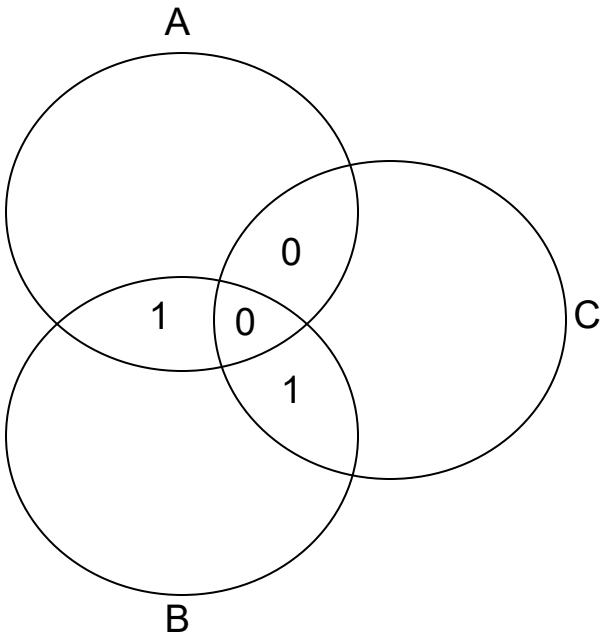
Error Correction

- The following is a simple scheme (called Hamming(7,4)) for encoding data such that errors are correctable
  - The scheme used by ECC RAM is similar
- In the left diagram, we have 4 bits of information encoded
  - In the centre diagram, the ‘parity’ bits have been added to make even parity within the circles
  - The parity overlaps, so that we can figure out which bit is in error
  - The right diagram shows an example with a data bit error (parity bit errors are similar)



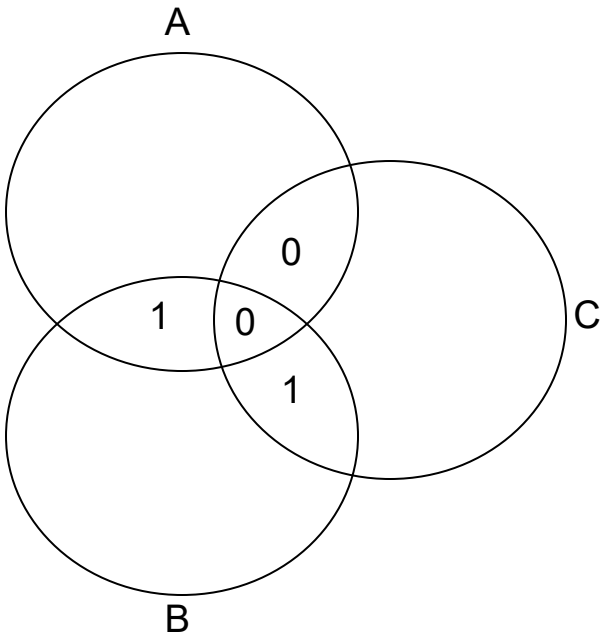
Error Correction

- Let’s do an example (even parity)
  - Consider the following bits: `1100`



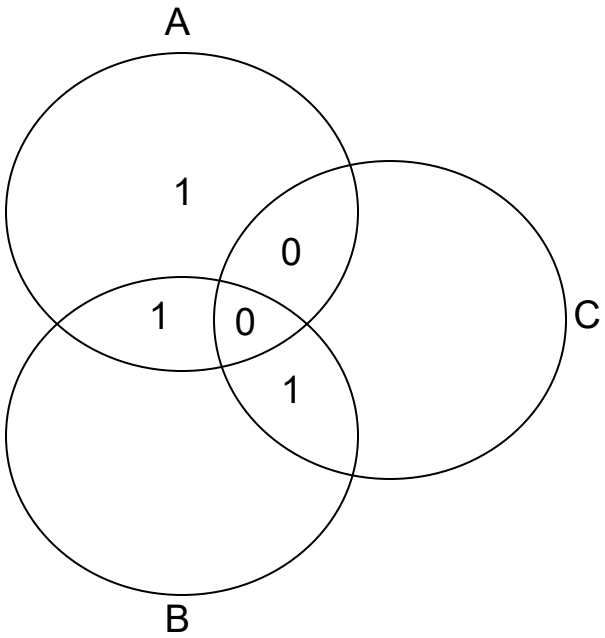
Error Correction

- Let’s do an example (even parity)
  - What should the parity bit be for A?



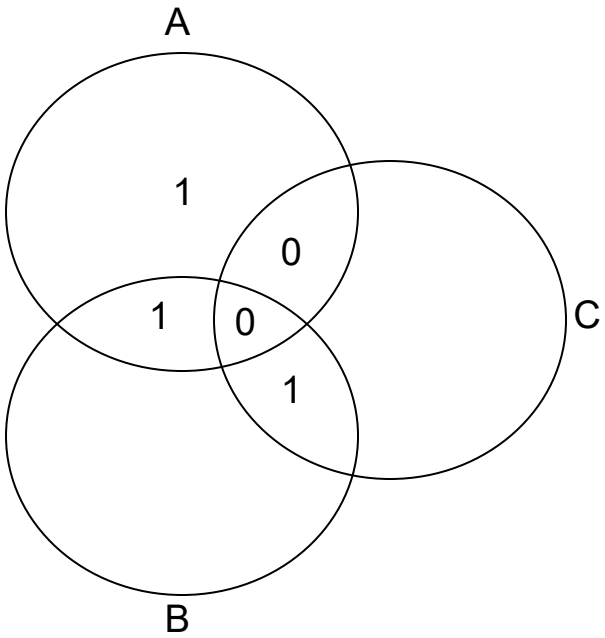
Error Correction

- Let’s do an example (even parity)
  - What should the parity bit be for A? `1`



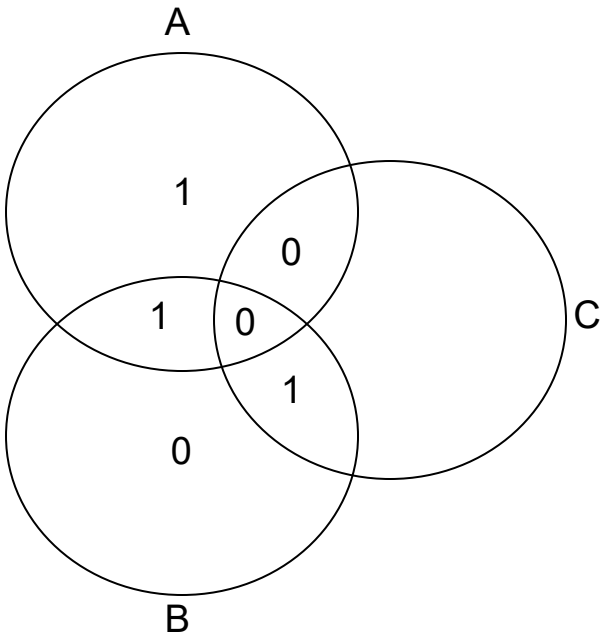
Error Correction

- Let’s do an example (even parity)
  - What should the parity bit be for B?



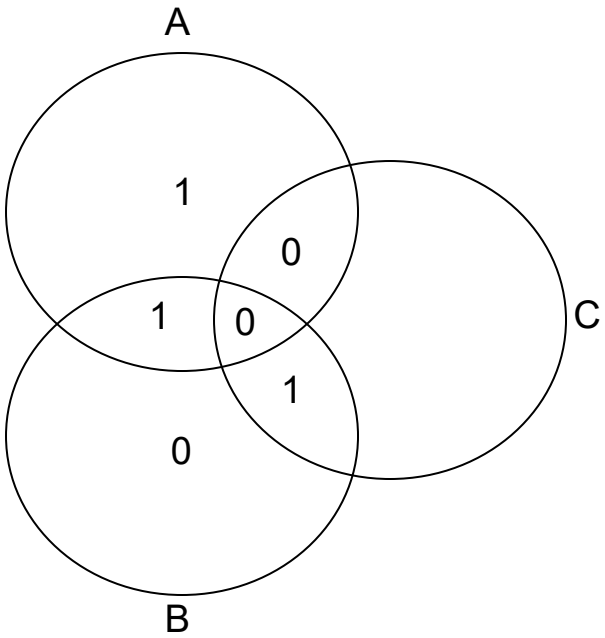
Error Correction

- Let’s do an example (even parity)
  - What should the parity bit be for B? 0



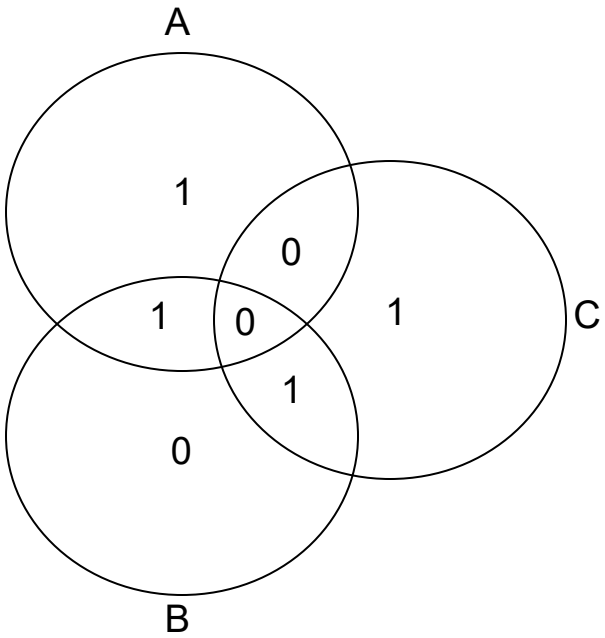
Error Correction

- Let’s do an example (even parity)
  - What should the parity bit be for C?



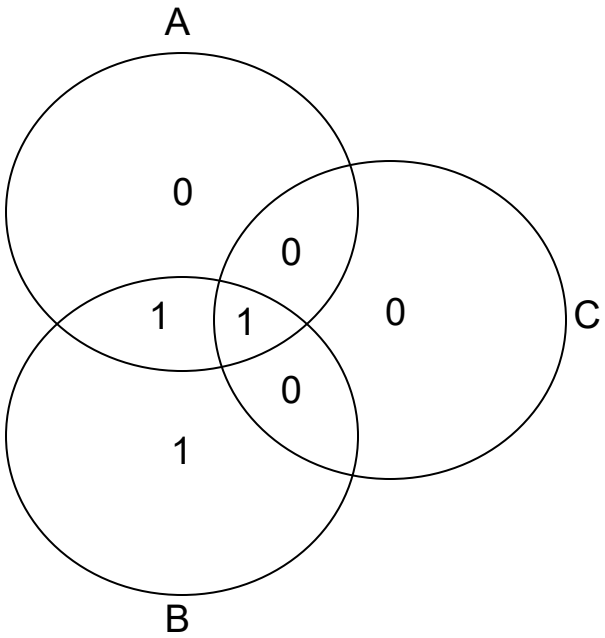
Error Correction

- Let’s do an example (even parity)
  - What should the parity bit be for C? `1`



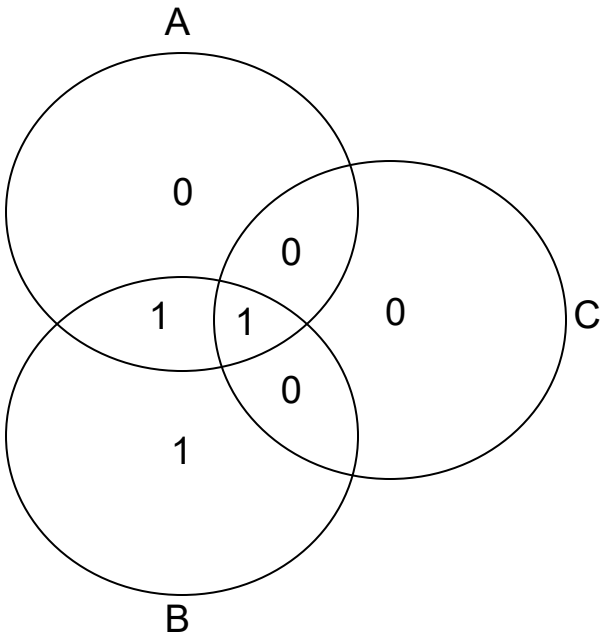
Error Correction

- Let’s do another example (even parity)
  - Let’s verify the parity bits:
    - A:



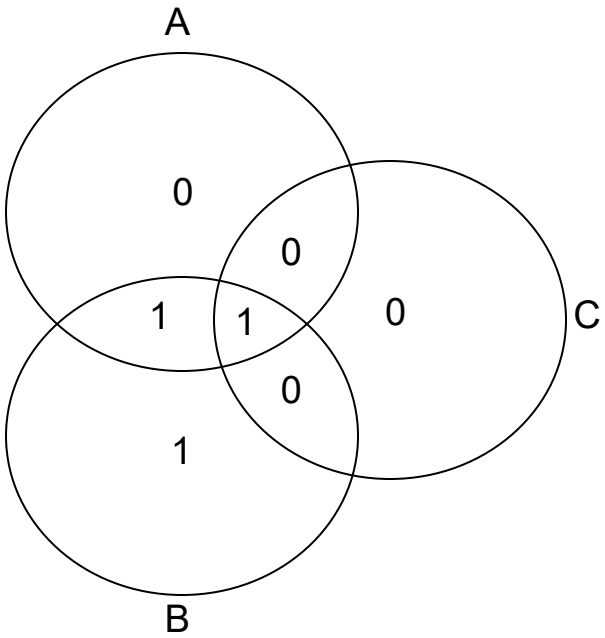
Error Correction

- Let’s do another example (even parity)
  - Let’s verify the parity bits:
    - A: correct
    - B:



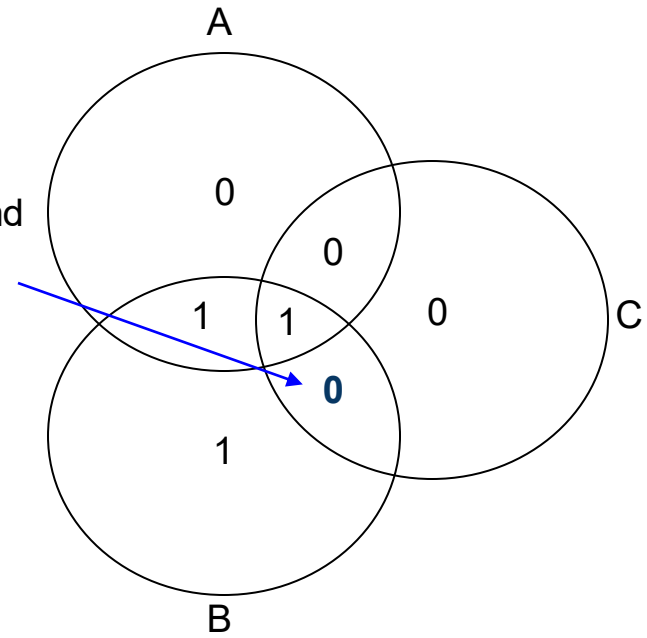
Error Correction

- Let’s do another example (even parity)
  - Let’s verify the parity bits:
    - A: correct
    - B: incorrect
    - C:



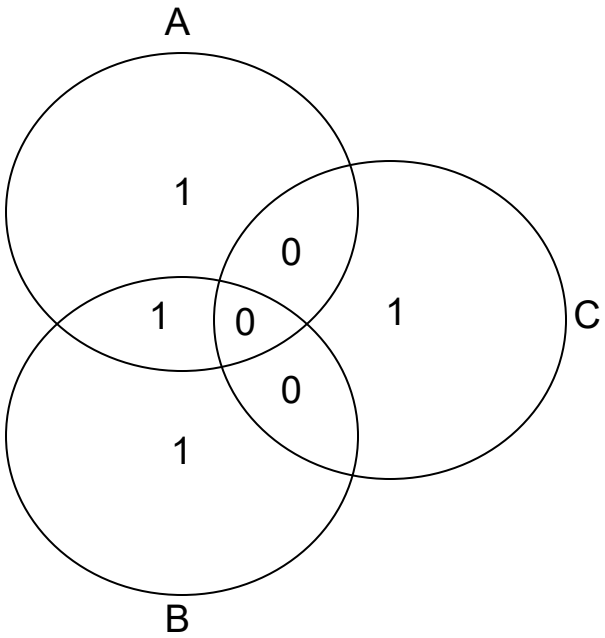
Error Correction

- Let's do another example (even parity)
  - Let's verify the parity bits:
    - A: correct
    - B: incorrect
    - C: incorrect
  - Since the error must be in both B and C, but not A, the error must be **here**



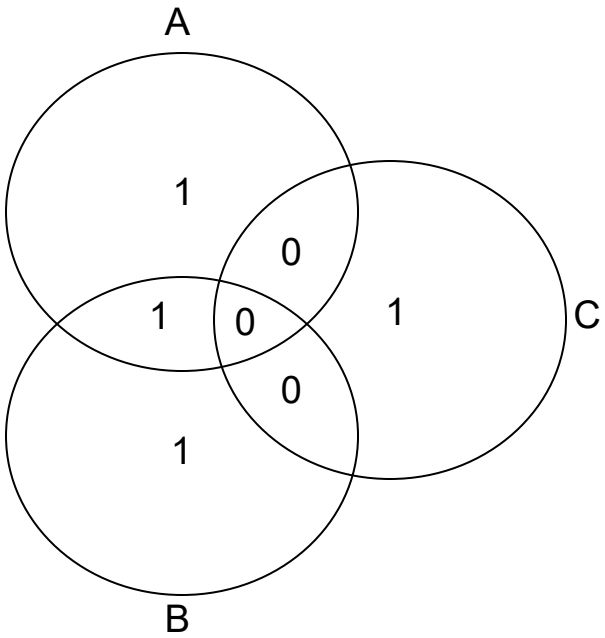
Error Correction

- Let’s do another example (even parity)
  - Let’s verify the parity bits:
    - A:



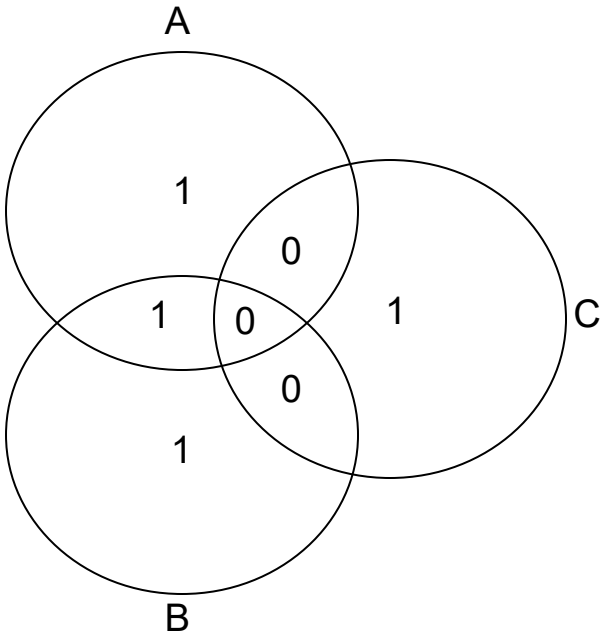
Error Correction

- Let’s do another example (even parity)
  - Let’s verify the parity bits:
    - A: correct
    - B:



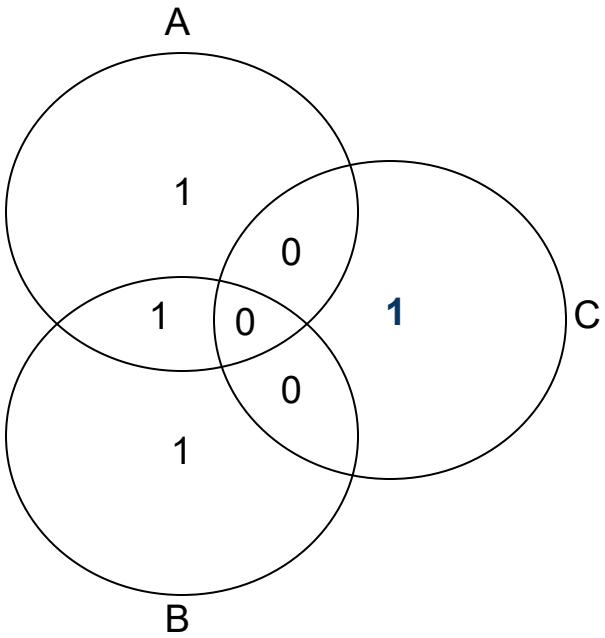
Error Correction

- Let's do another example (even parity)
  - Let's verify the parity bits:
    - A: correct
    - B: correct
    - C:



Error Correction

- Let's do another example (even parity)
  - Let's verify the parity bits:
    - A: correct
    - B: correct
    - C: incorrect
  - Since the error must be in C, but not in A or B, the error must be the parity bit in C
  - The parity bits even check themselves!



Error Correction

- Extending this system to larger values is a bit more complicated
- The overhead (extra bits required) becomes less significant for larger values

Word size	Check bits	Total size	Percent overhead
8	4	12	50
16	5	21	31
32	6	38	19
64	7	71	11
128	8	136	6
256	9	265	4
512	10	522	2

See: https://www.youtube.com/watch?v=X8jsijhllIA and https://www.youtube.com/watch?v=b3NxrZOu_CE for more information

Encoding

CSCI 2050U - Computer Architecture

OntarioTech UNIVERSITY

Fixed-length Encoding

- Say we want to encode DNA in binary
  - DNA is made up of sequences of nucleotides
  - There are four possible values (see table)
  - For simplicity, we'll say that each is equally likely

A	adenine
C	cytosine
G	guanine
T	thymine

Fixed-length Encoding

- Say we want to encode DNA in binary
  - DNA is made up of sequences of nucleotides
  - There are four possible values (see table)
  - For simplicity, we'll say that each is equally likely
- Other fixed-length encodings:
  - ASCII

A	adenine	00
C	cytosine	01
G	guanine	10
T	thymine	11

Variable-length Encoding

- What if the choices are *not* equally likely?
- Consider letters in the English alphabet (e.g. encoding a text message)
  - We could save some space if more common letters were given shorter codes
- Let's say that 'a' and 'b' are common, but 'c' and 'd' are less common
  - What about the following codes?

Letter	***Code***
a	0
b	1
c	00
d	11

Variable-length Encoding

- Let's consider the binary value: `000111`
- Possible interpretations:
  - `aaabbb`
  - `acbd`
  - `cabd`
  - `acdb`
  - `cadb`
  - `aaabd`
  - `aaadb`
  - `acbbb`
  - `cabbb`

Letter	***Code***
a	0
b	1
c	00
d	11

Variable-length Encoding

- In order to avoid ambiguity, we need the codes to be prefix codes
  - In a prefix code, each code has a unique prefix that is not shared with any other code
  - In this way, we can identify each encoded symbol
- Let's consider the binary value: `110100101`
  - The only interpretation of this binary value is `badc`

Letter	***Code***
a	0
b	11
c	101
d	100

Huffman's Algorithm

```
1. HUFFMAN(C):
2. n = |C|
3. Q = C
4. for i = 1 to n-1
5.     create a new node z
6.     x = EXTRACT-MIN(Q)
7.     y = EXTRACT-MIN(Q)
8.     z.left = x
9.     z.right = y
10.    z.freq = x.freq + y.freq
11.    INSERT(Q,z)
12. end for
13. return EXTRACT-MIN(Q)
```

Huffman’s Algorithm

- Huffman’s algorithm uses a (min) priority queue
  - A priority queue behaves like a queue, except that higher priority items jump ahead of lower priority items
    - This is similar to triage in an emergency room

Character	***Frequency***
a	45
b	13
c	12
d	16
e	9
f	5

Huffman's Algorithm

- We start by adding all characters/frequencies to a min priority queue
  - Frequencies, in this case, are the priority value

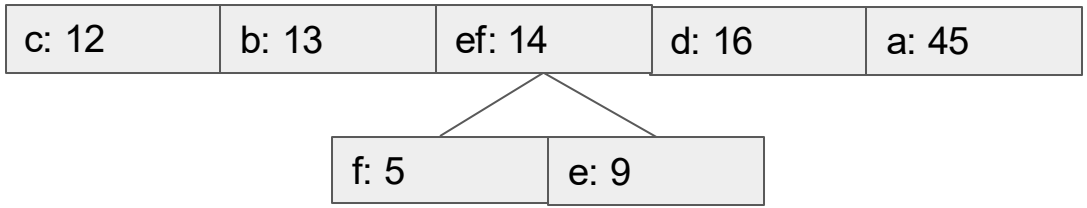
Character	*Frequency*
a	45
b	13
c	12
d	16
e	9
f	5

f: 5	e: 9	c: 12	b: 13	d: 16	a: 45

Huffman's Algorithm

- Dequeue the first two elements from the priority queue
  - Join them together into a single node and enqueue that new node
  - The frequency of the new node is the sum of the frequencies of the two elements dequeued

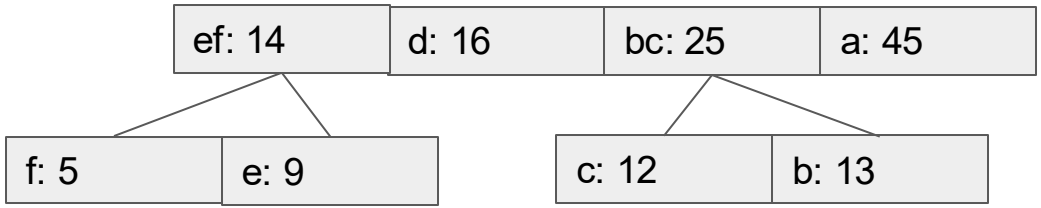
Character	*Frequency*
a	45
b	13
c	12
d	16
e	9
f	5



Huffman's Algorithm

- Keep repeating this process until there is only a single node in the queue:

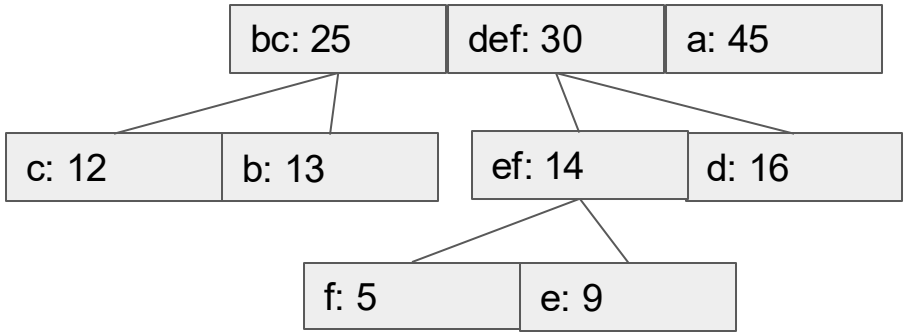
Character	*Frequency*
a	45
b	13
c	12
d	16
e	9
f	5



Huffman's Algorithm

- Keep repeating this process until there is only a single node in the queue:

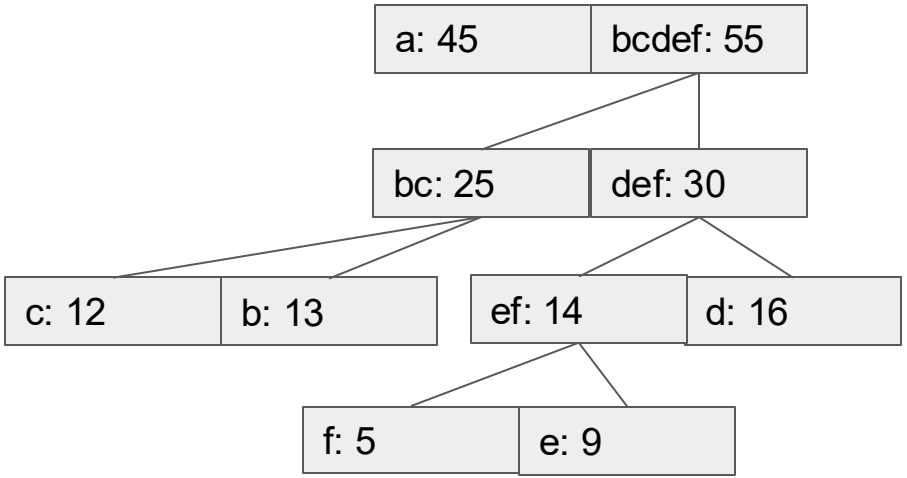
Character	*Frequency*
a	45
b	13
c	12
d	16
e	9
f	5



Huffman's Algorithm

- Keep repeating this process until there is only a single node in the queue:

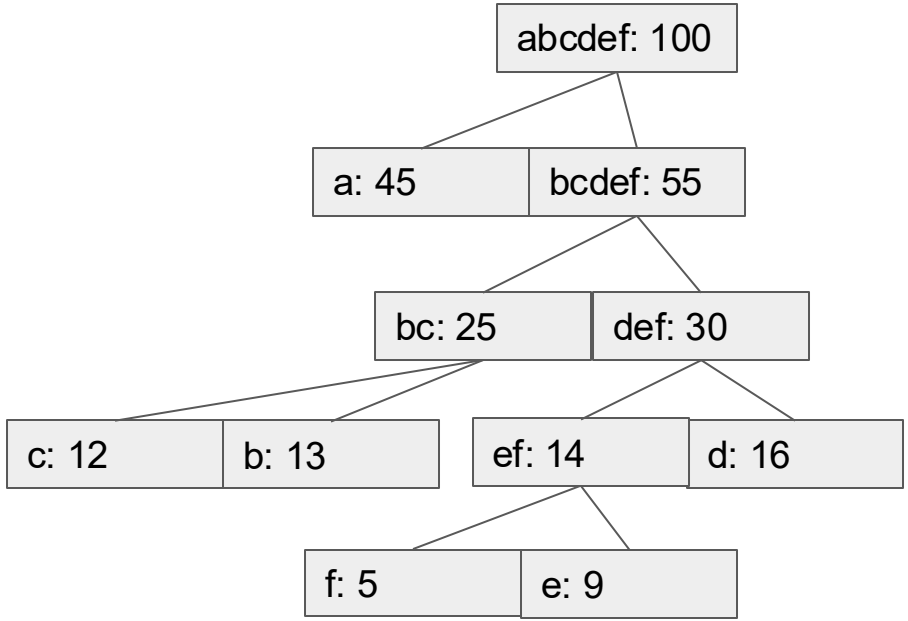
Character	*Frequency*
a	45
b	13
c	12
d	16
e	9
f	5



Huffman's Algorithm

- Keep repeating this process until there is only a single node in the queue:

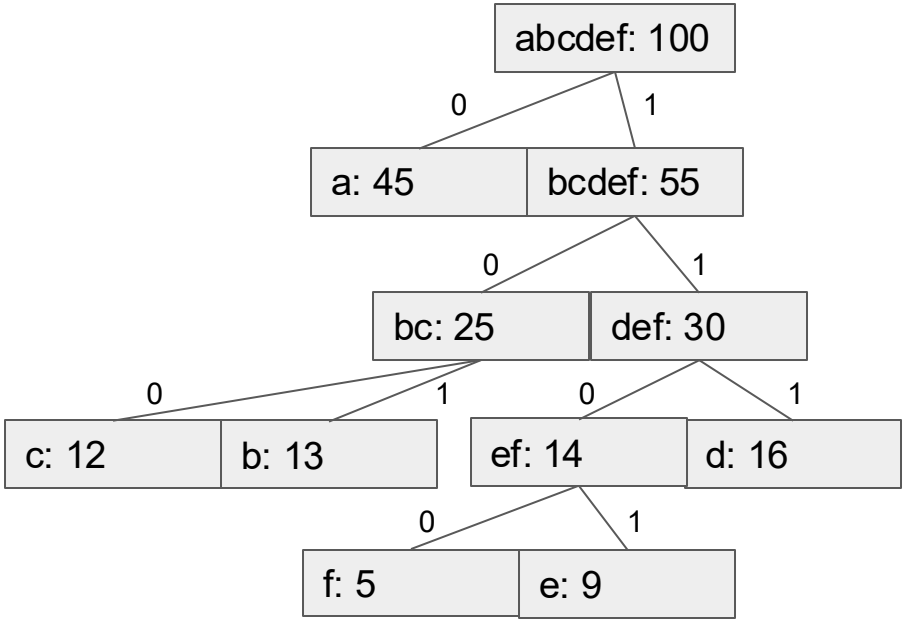
Character	*Frequency*
a	45
b	13
c	12
d	16
e	9
f	5



Huffman's Algorithm

- Now label each left child 0, and each right child 1
  - The prefix codes for each letter can be found by tracing the path to that node in this tree

Character	*Frequency*	*Prefix Code*
a	45	0
b	13	101
c	12	100
d	16	111
e	9	1101
f	5	1100



Wrap-up

- Errors detection and correction
  - Parity
  - Hamming distance
  - Error correcting codes
- Encoding
  - Prefix codes
  - Huffman's algorithm

What is next?

- Basic electronics
  - Diodes
    - Light emitting diodes (LEDs)
  - Resistors
  - Transistors