10-0: Text Files

- All files are represented as binary digits – including text files
- Each character is represented by an integer code
    - ASCII – American Standard Code for Information Interchange
- Text file is a sequence of binary digits which represent the codes for each character.

10-1: ASCII

- Each character can be represented as an 8-bit number
    - ASCII for a = 97 = 01100001
    - ASCII for b = 98 = 01100010
- Text file is a sequence of 1’s and 0’s which represent ASCII codes for characters in the file
    - File “aba” is 97, 97, 98
    - 011000010110001001100001

10-2: ASCII

- Each character in ASCII is represented as 8 bits
    - We need 8 bits to represent all possible character combinations
        - (including control characters, and unprintable characters)
    - Breaking up file into individual characters is easy
    - Finding the kth character in a file is easy

10-3: ASCII

- ASCII is not terribly efficient
    - All characters require 8 bits
    - Frequently used characters require the same number of bits as infrequently used characters
    - We could be more efficient if frequently used characters required fewer than 8 bits, and less frequently used characters required more bits

10-4: Representing Codes as Trees

- Want to encode 4 only characters: a, b, c, d (instead of 256 characters)
    - How many bits are required for each code, if each code has the same length?

10-5: Representing Codes as Trees

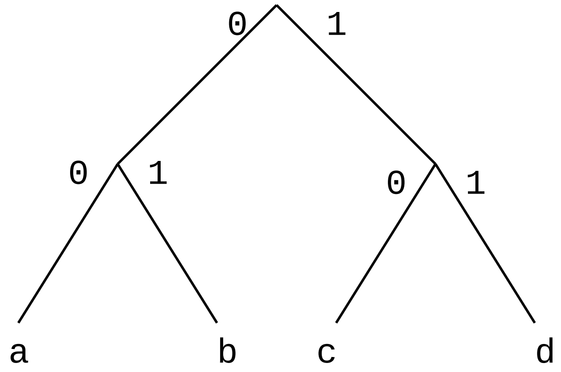
- Want to encode 4 only characters: a, b, c, d (instead of 256 characters)
    - How many bits are required for each code, if each code has the same length?
    - 2 bits are required, since there are 4 possible options to distinguish

10-6: Representing Codes as Trees

- Want to encode 4 only characters: a, b, c, d
- Pick the following codes:
  - a: 00
  - b: 01
  - c: 10
  - d: 11
- We can represent these codes as a tree
  - Characters are stored at the leaves of the tree
  - Code is represented by path to leaf

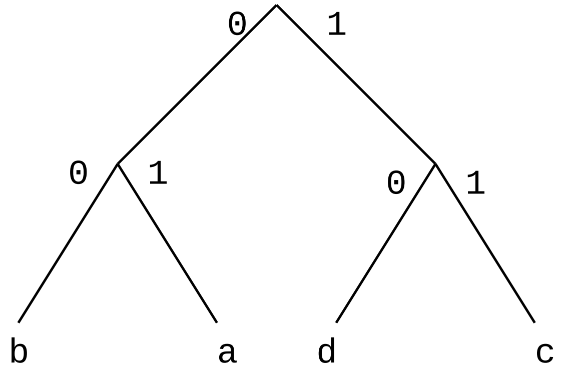
10-7: **Representing Codes as Trees**

- a: 00, b: 01, c: 10, d:11



10-8: **Representing Codes as Trees**

- a: 01, b: 00, c: 11, d:10



10-9: **Prefix Codes**

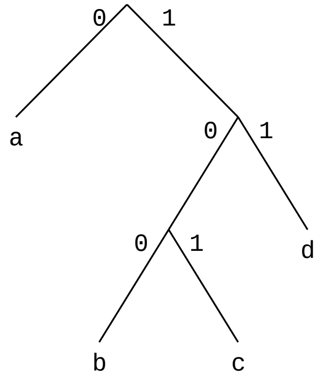
- If no code is a prefix of any other code, then decoding the file is unambiguous.
- If all codes are the same length, then no code will be a prefix of any other code (trivially)
- We can create variable length codes, where no code is a prefix of any other code

10-10: **Variable Length Codes**

- Variable length code example:
    - a: 0, b: 100, c: 101, d: 11
- Decoding examples:
    - 100
    - 10011
    - 01101010010011

10-11: **Prefix Codes & Trees**

- Any prefix code can be represented as a tree
- a: 0, b: 100, c: 101, d: 11



10-12: **File Length**

- If we use the code:
    - a:00, b:01, c:10, d:11

  How many bits are required to encode a file of 20 characters?

10-13: **File Length**

- If we use the code:
    - a:00, b:01, c:10, d:11

  How many bits are required to encode a file of 20 characters?
- 20 characters * 2 bits/character = 40 bits

10-14: **File Length**

- If we use the code:
    - a:0, b:100, c:101, d:11

  How many bits are required to encode a file of 20 characters?

10-15: **File Length**

- If we use the code:
  - a:0, b:100, c:101, d:11

  How many bits are required to encode a file of 20 characters?
- It depends upon the number of a's, b's, c's and d's in the file

10-16: **File Length**

- If we use the code:
  - a:0, b:100, c:101, d:11

  How many bits are required to encode a file of:
  - 11 a's, 2 b's, 2 c's, and 5 d's?

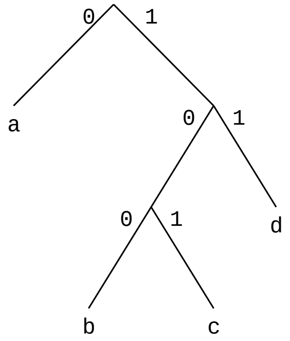
10-17: **File Length**

- If we use the code:
  - a:0, b:100, c:101, d:11

  How many bits are required to encode a file of:
  - 11 a's, 2 b's, 2 c's, and 5 d's?
- 11*1 + 2*3 + 2*3 + 5*2 = 33 ¡ 40

10-18: **Decoding Files**

- We can use variable length keys to encode a text file
- Given the encoded file, and the tree representation of the codes, it is easy to decode the file



- 0111001010011

10-19: **Decoding Files**

- We can use variable length keys to encode a text file
- Given the encoded file, and the tree representation of the codes, it is easy to decode the file
- Finding the kth character in the file is more tricky

10-20: **Decoding Files**

- We can use variable length keys to encode a text file
- Given the encoded file, and the tree representation of the codes, it is easy to decode the file
- Finding the kth character in the file is more tricky
    - Need to decode the first (k-1) characters in the file, to determine where the kth character is in the file

10-21: **File Compression**

- We can use variable length codes to compress files
    - Select an encoding such that frequently used characters have short codes, less frequently used characters have longer codes
    - Write out the file using these codes
    - (If the codes are dependent upon the contents of the file itself, we will also need to write out the codes at the beginning of the file for decoding)

10-22: **File Compression**

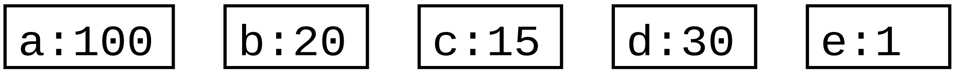
- We need a method for building codes such that:
    - Frequently used characters are represented by leaves high in the code tree
    - Less Frequently used characters are represented by leaves low in the code tree
    - Characters of equal frequency have equal depths in the code tree

10-23: **Huffman Coding**

- For each code tree, we keep track of the total number of times the characters in that tree appear in the input file
- We start with one code tree for each character that appears in the input file
- We combine the two trees with the lowest frequency, until all trees have been combined into one tree

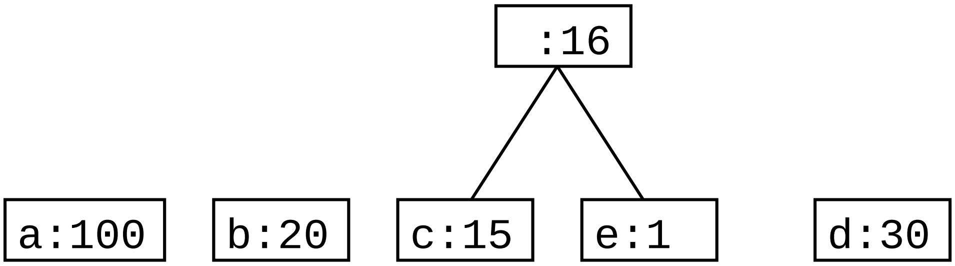
10-24: **Huffman Coding**

- Example: If the letters a-e have the frequencies:
    - a: 100, b: 20, c:15, d: 30, e: 1



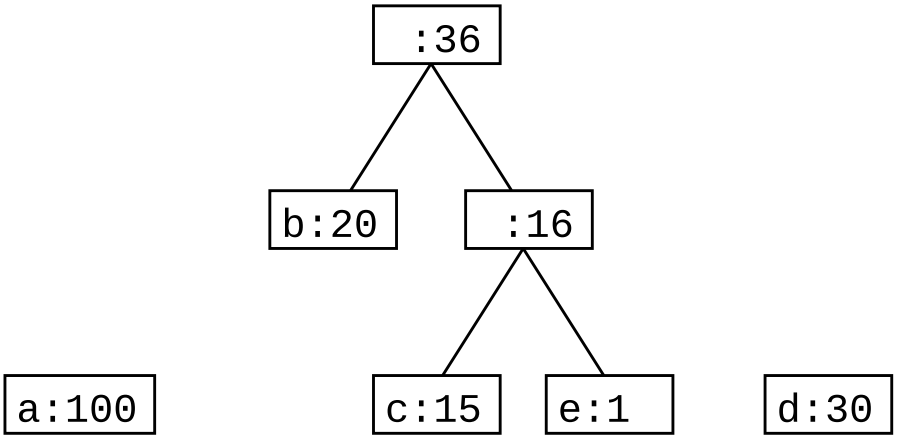
10-25: **Huffman Coding**

- Example: If the letters a-e have the frequencies:
    - a: 100, b: 20, c:15, d: 30, e: 1



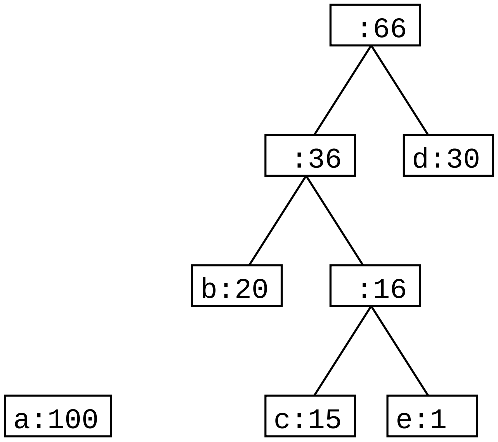
10-26: **Huffman Coding**

- Example: If the letters a-e have the frequencies:
  - a: 100, b: 20, c:15, d: 30, e: 1



10-27: **Huffman Coding**

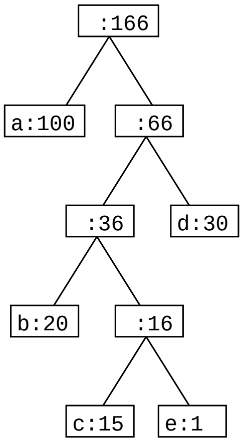
- Example: If the letters a-e have the frequencies:
  - a: 100, b: 20, c:15, d: 30, e: 1



10-28: **Huffman Coding**

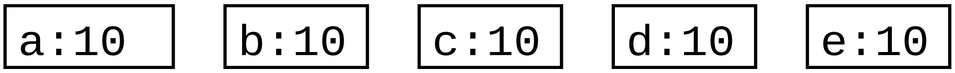
- Example: If the letters a-e have the frequencies:

- a: 100, b: 20, c:15, d: 30, e: 1



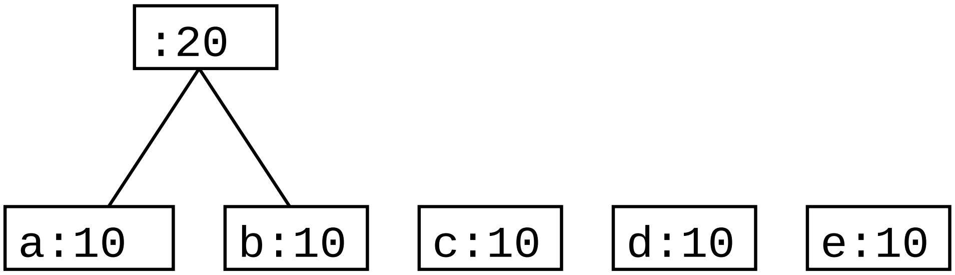
10-29: **Huffman Coding**

- Example: If the letters a-e have the frequencies:
  - a: 10, b: 10, c:10, d: 10, e: 10



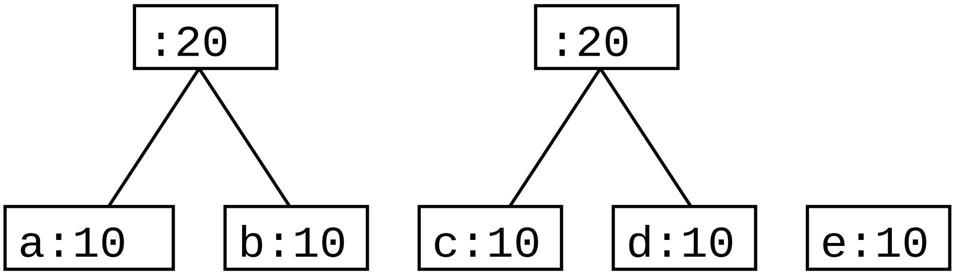
10-30: **Huffman Coding**

- Example: If the letters a-e have the frequencies:
  - a: 10, b: 10, c:10, d: 10, e: 10



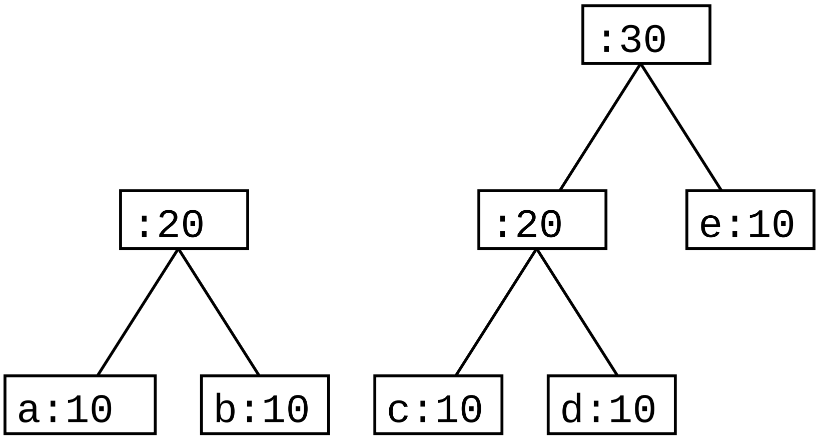
10-31: **Huffman Coding**

- Example: If the letters a-e have the frequencies:
  - a: 10, b: 10, c:10, d: 10, e: 10



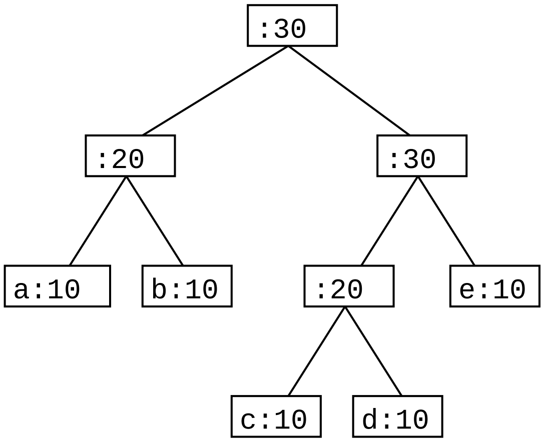
10-32: **Huffman Coding**

- Example: If the letters a-e have the frequencies:
  - a: 10, b: 10, c:10, d: 10, e: 10



10-33: **Huffman Coding**

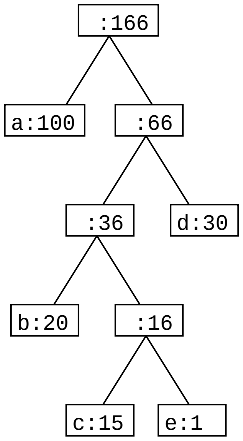
- Example: If the letters a-e have the frequencies:
  - a: 10, b: 10, c:10, d: 10, e: 10



10-34: **Huffman Trees & Tables**

- Once we have a Huffman tree, decoding a file is straightforward – but *encoding* a tree requires a bit more information.
- Given just the tree, finding an encoding can be difficult
- ... What would we like to have, to help with encoding?

10-35: **Encoding Tables**



a	0
b	100
c	1010
d	11
e	1011

10-36: **Creating Encoding Table**

- Traverse the tree
  - Keep track of the path during the traversal
- When a leaf is reached, store the path in the table

10-37: **Huffman Coding**

- To compress a file using huffman coding:
  - Read in the file, and count the occurrence of each character, and built a frequency table
  - Build the Huffman tree from the frequencies
  - Build the Huffman codes from the tree
  - Print the Huffman tree to the output file (for use in decompression)
  - Print out the codes for each character

10-38: **Huffman Coding**

- To uncompress a file using huffman coding:
  - Read in the Huffman tree from the input file
  - Read the input file bit by bit, traversing the Huffman tree as you go
  - When a leaf is read, write the appropriate file to an output file

10-39: **Printing out Trees**

- To print out Huffman trees:
  - Print out nodes in pre-order traversal
  - Need a way of denoting which nodes are leaves and which nodes are interior nodes
    - (Huffman trees are full – every node has 0 or 2 children)
  - For each interior node, print out a 0 (single bit). For each leaf, print out a 1, followed by 8 bits for the character at the leaf

10-40: **Compression?**

- Is it possible that huffman compression would not compress the file?
- Is it possible that huffman compression could actually make the file larger?
- How?

10-41: **Compression?**

- What happens if all the charcters have the same frequency?
  - What does the tree look like?
  - What can we say about the lengths of the codes for each character?
  - What does that mean for the file size?

10-42: **Compression?**

- What happens if all the charcters have the same frequency?
  - All nodes are at the same depth in the tree (that is, 8)
  - Each code will have a length of 8
  - The encoded file will be the same size as the original file – *plus the size required to encode the tree*