Data Structures and Algorithms
Huffman Trees

Chris Brooks

Department of Computer Science

University of San Francisco

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 7 bits to represent all possible character combinations
  - The 8th bit is used for error correction.
  - 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

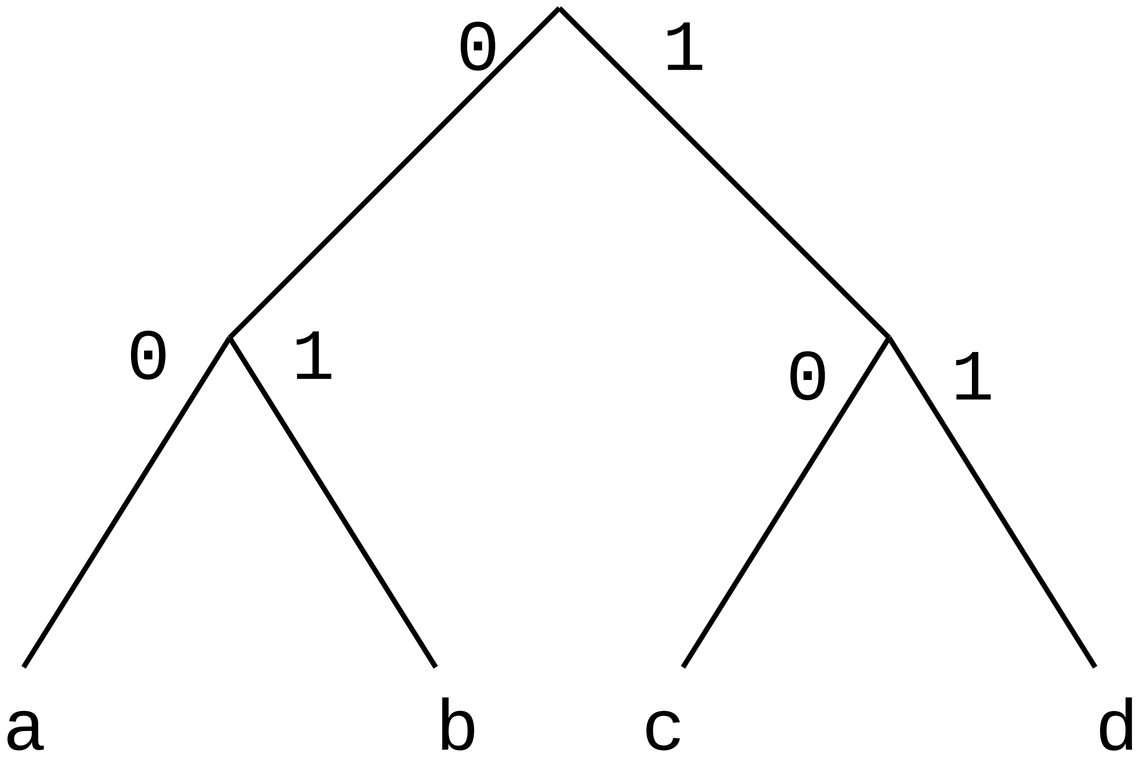
- Want to encode 4 only characters: a, b, c, d (instead of 256 characters)
  - How many bits are required for each code, if each code has the same length?
  - 2 bits are required, since there are 4 possible options to distinguish

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

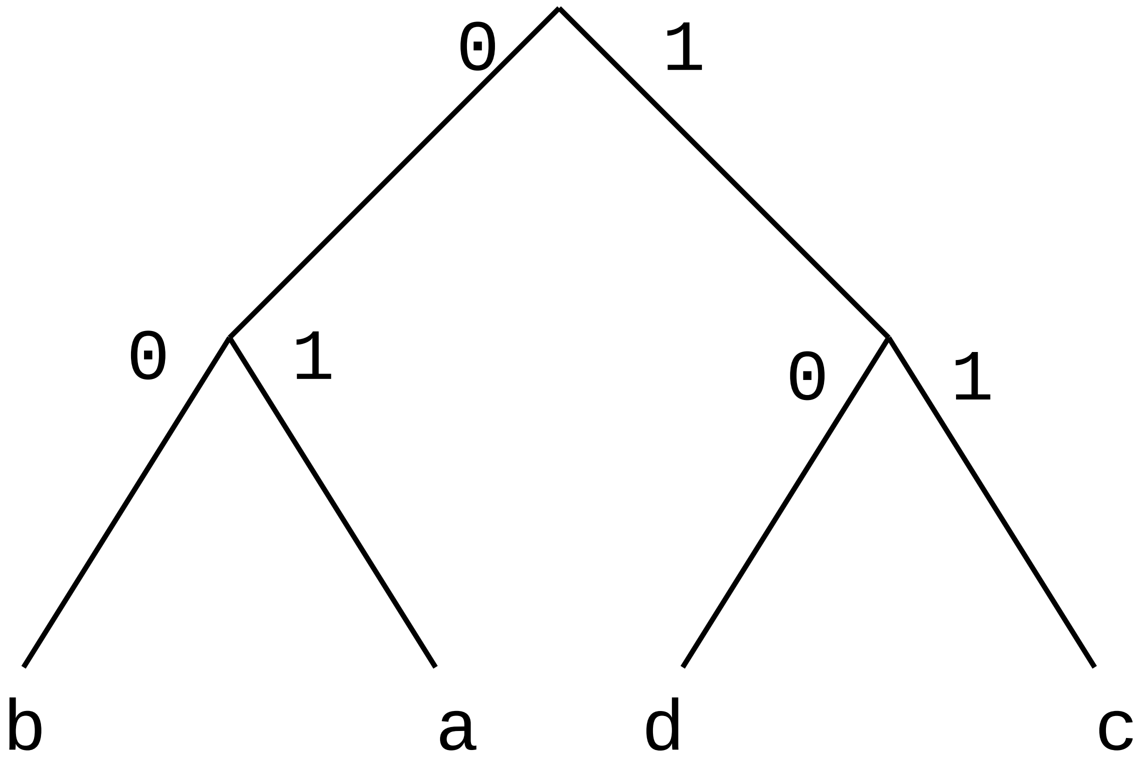
10-7: Representing Codes as Trees

- 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

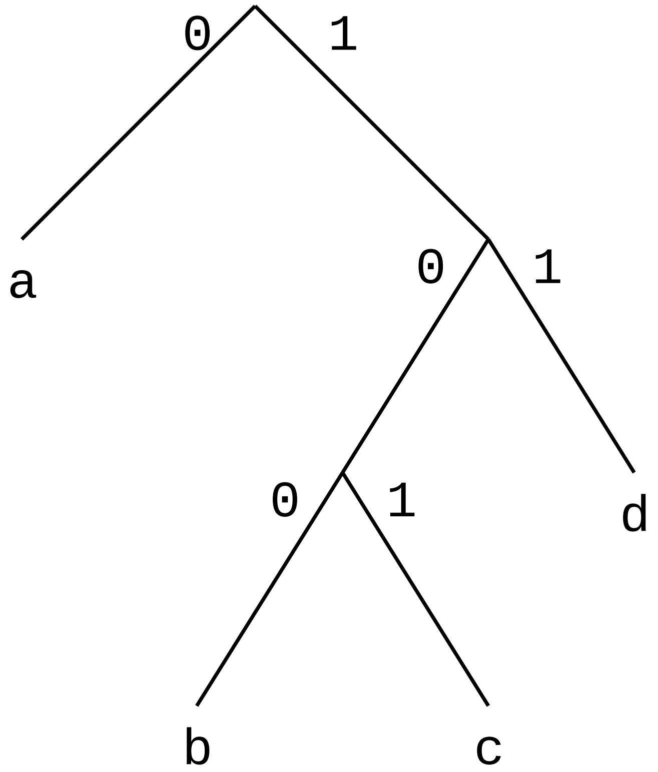
- If no code is a prefix of any other code, then decoding the file is unambiguous.
  - How do you know whether a string is one complete code, or part of another?
- If all codes are the same length, then no code will be a prefix of any other code (trivially)
- We can create variable length codes, where 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?

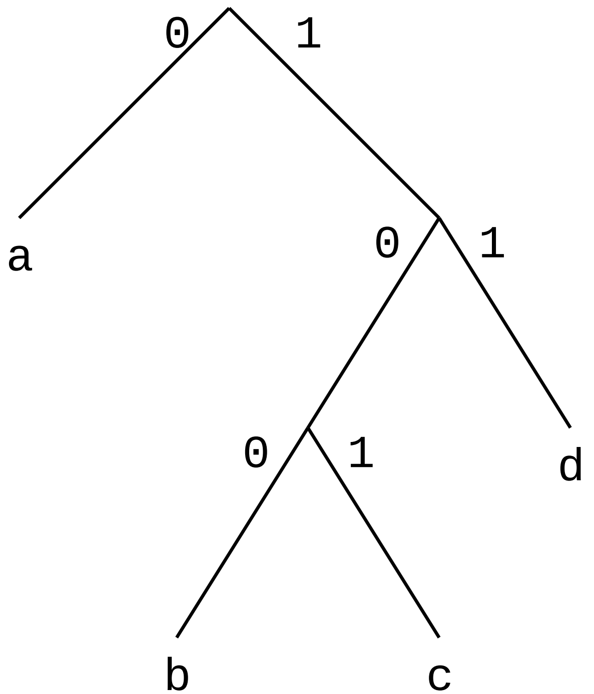
10-17: File Length

- 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
  - Gain space, lose random access.

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)

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

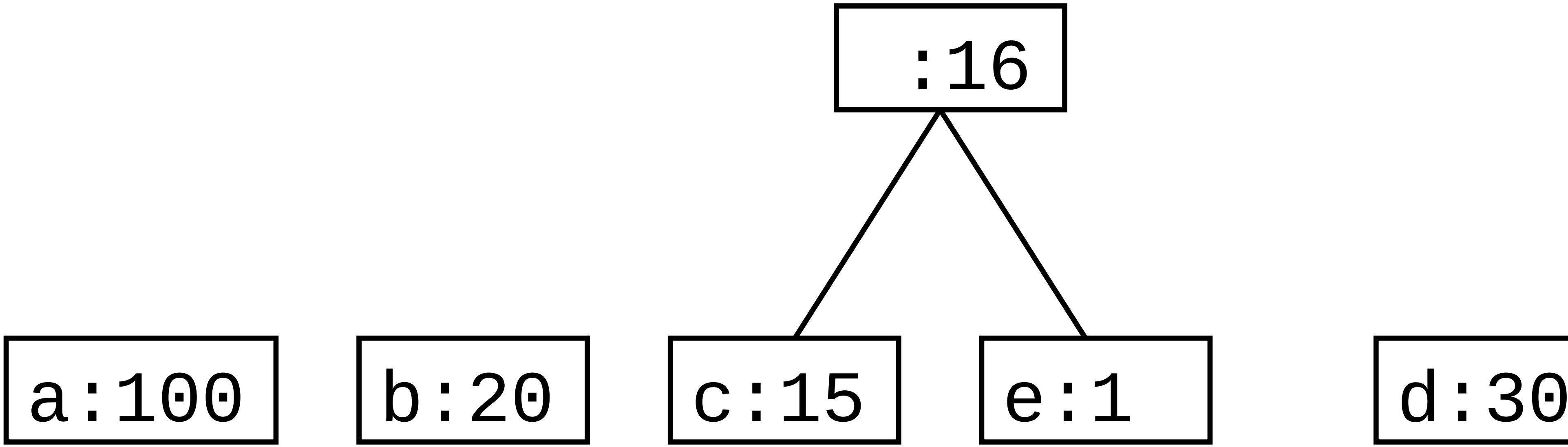
10-24: Huffman Coding

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

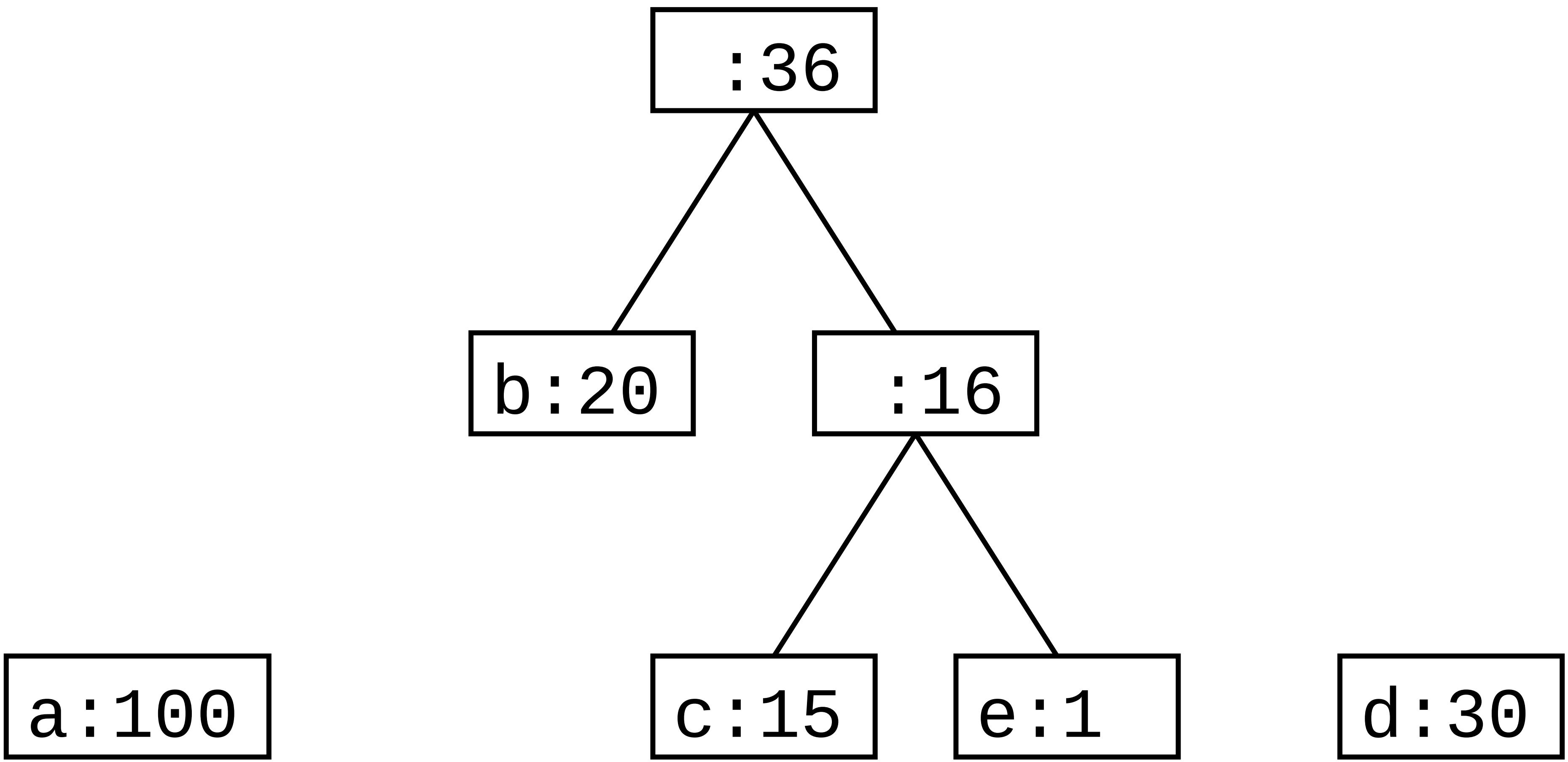
a:100 b:20 c:15 d:30 e:1

10-25: Huffman Coding

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

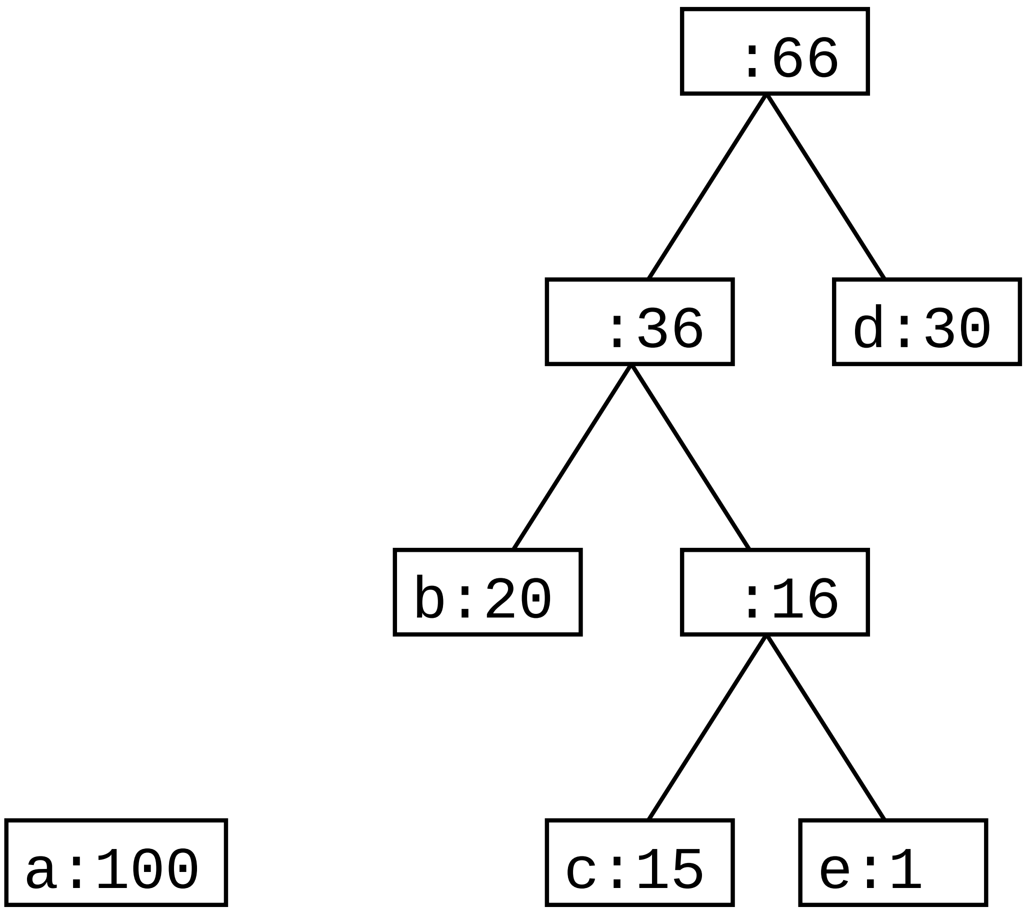


- Example: If the letters a-e have the frequencies:
  - a: 100, b: 20, c:15, d: 30, e: 1

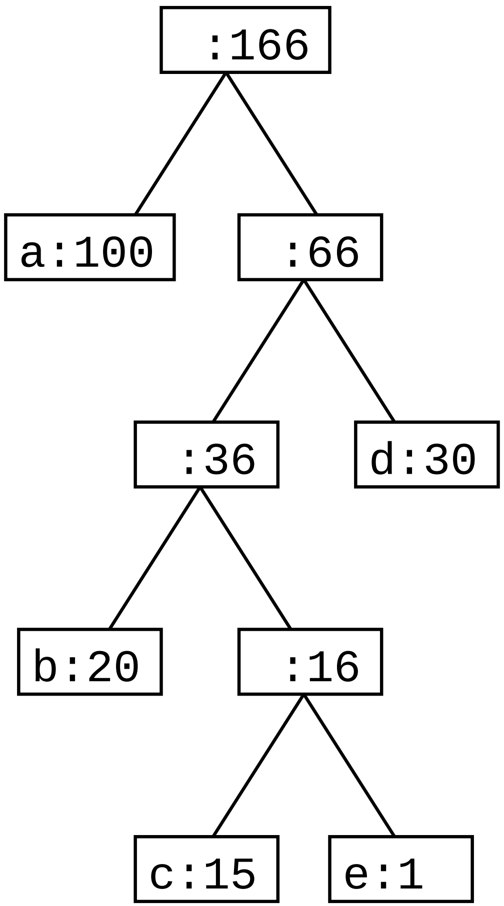


10-27: Huffman Coding

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



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



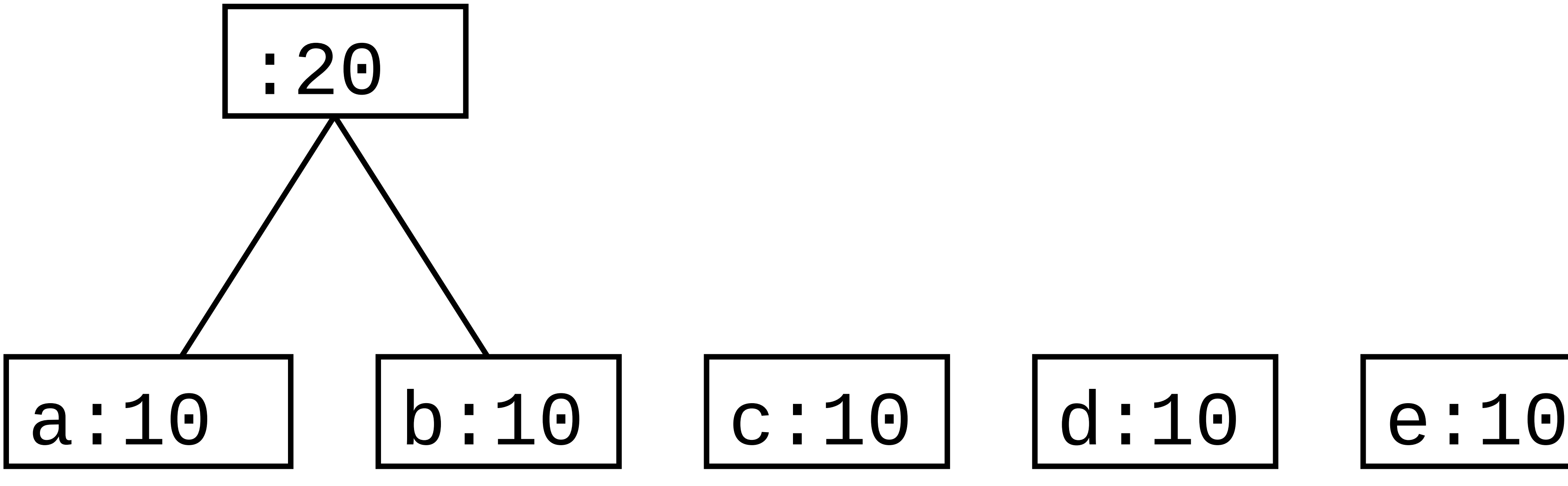
10-29: Huffman Coding

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

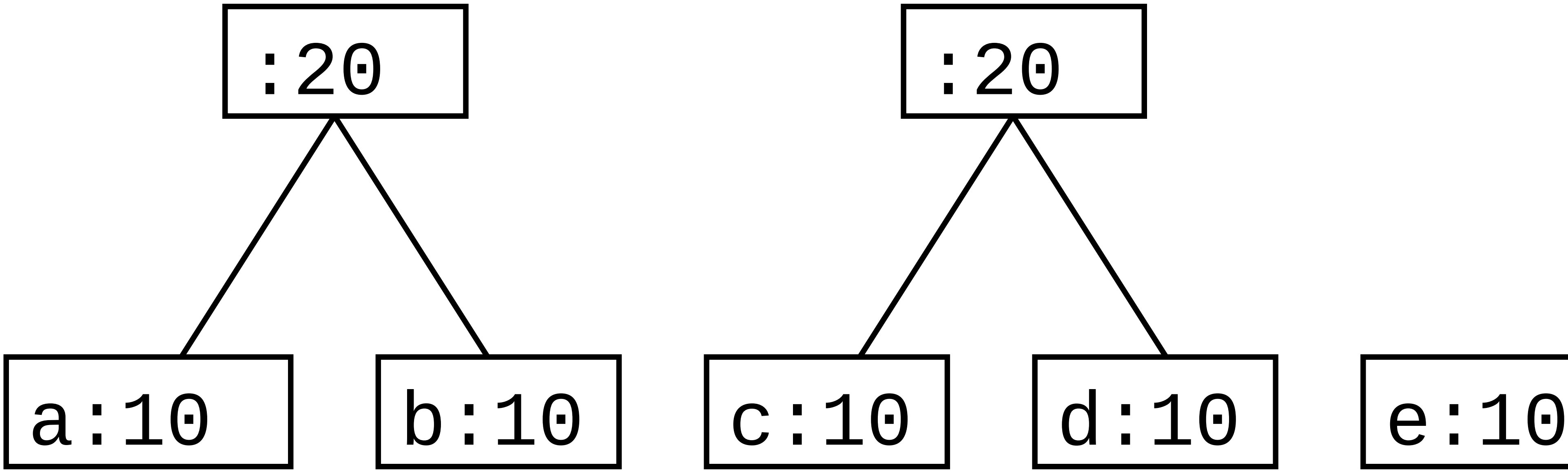
a:10	b:10	c:10	d:10	e:10

10-30: Huffman Coding

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

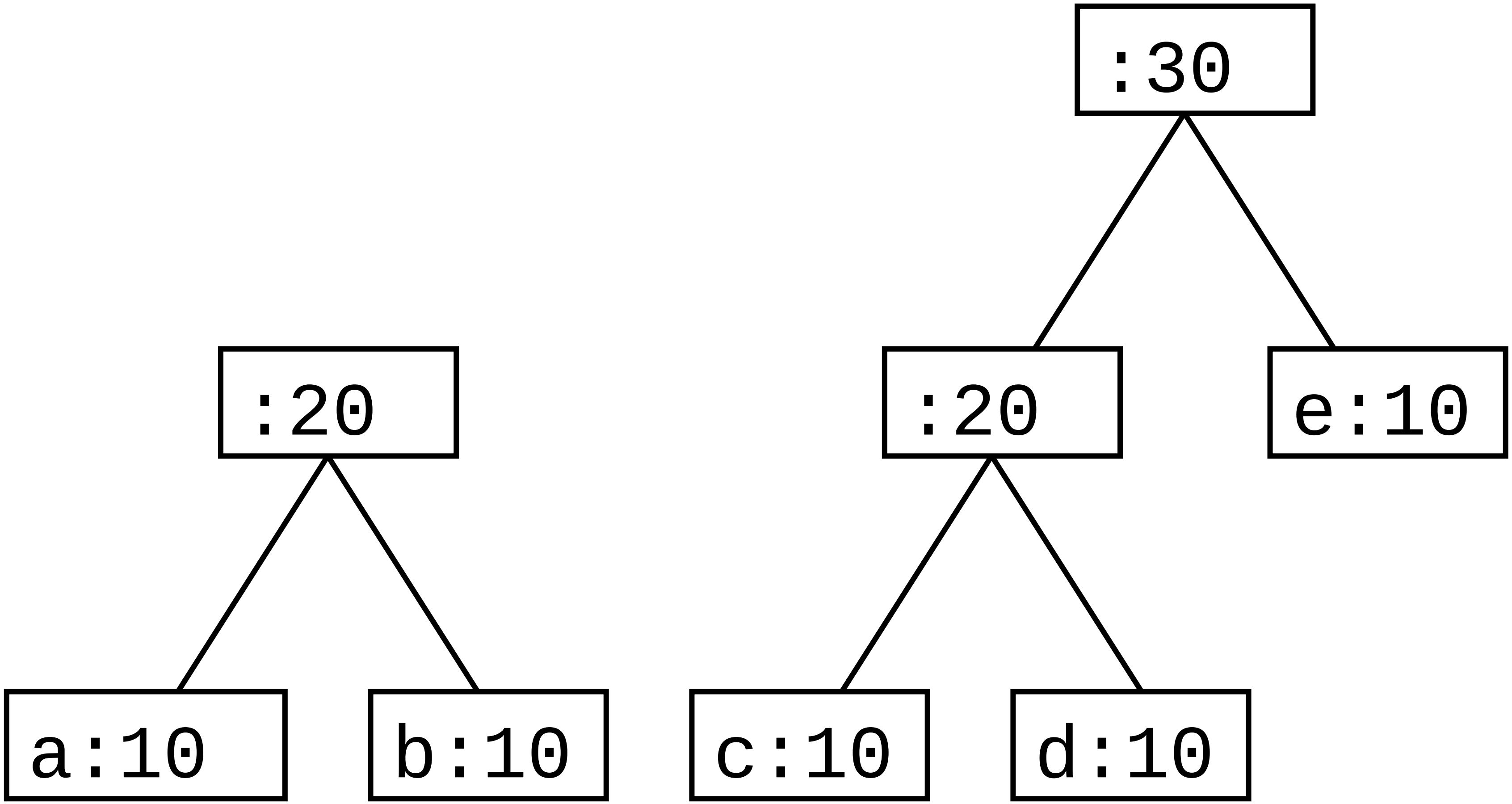


- Example: If the letters a-e have the frequencies:
  - a: 10, b: 10, c:10, d: 10, e: 10

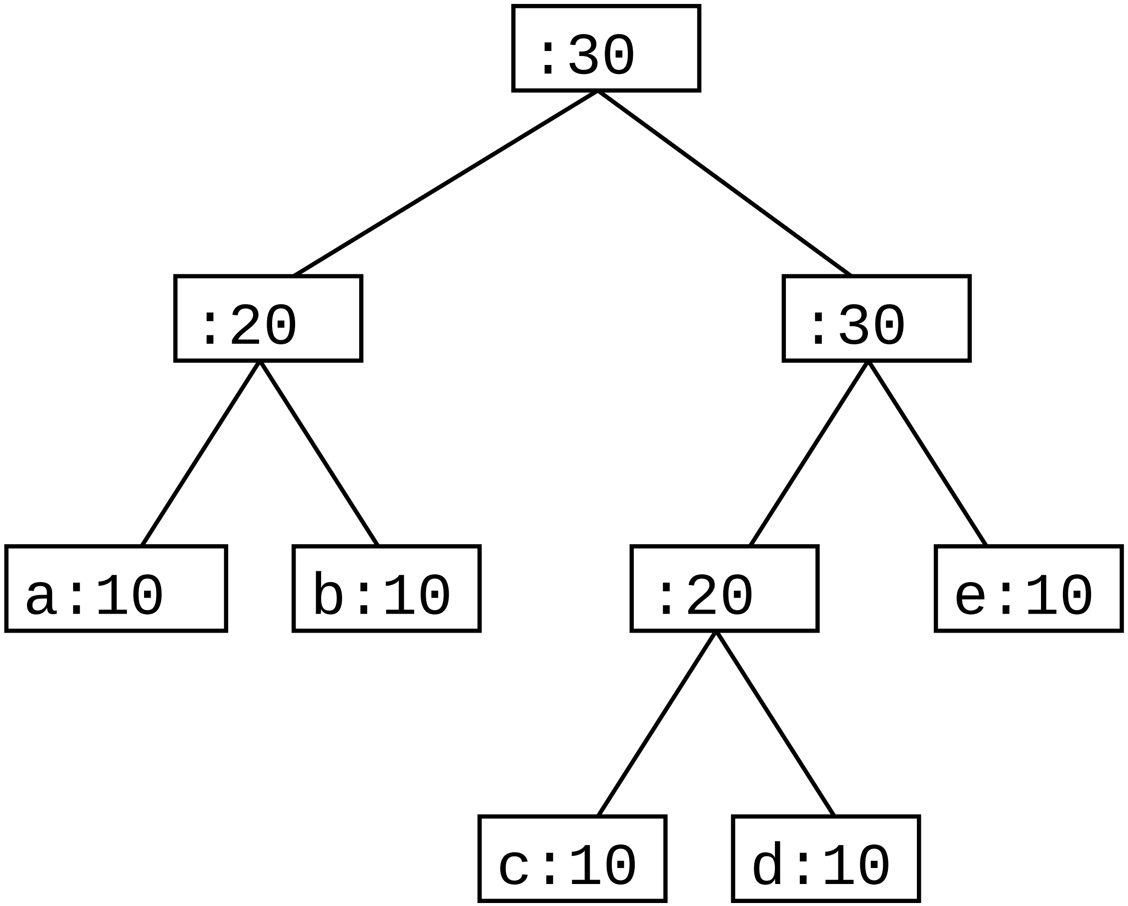


10-32: Huffman Coding

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



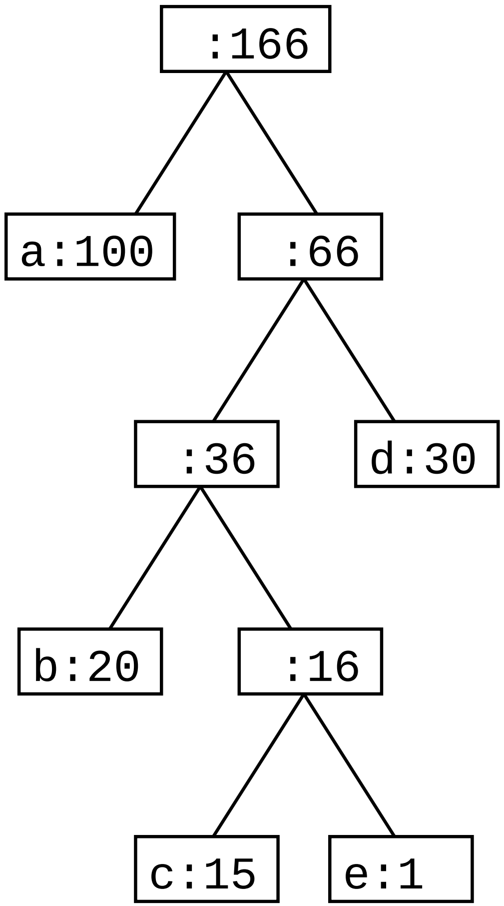
- Example: If the letters a-e have the frequencies:
  - a: 10, b: 10, c:10, d: 10, e: 10



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: Binary Files

```
public BinaryFile(String filename,
                  char readOrWrite)

public boolean EndOfFile()
public char readChar()
public void writeChar(char c)

public int readInt()
public void writeInt(int i)

public boolean readBit()
public void writeBit(boolean bit)

public void close()
```

10-40: Binary Files

- readBit
  - Read a single bit
- readChar
  - Read a single character (8 bits)
- readInt
  - Read a single int (9 bits) in the range -255 . . . 255

10-41: Binary Files

- writeBit
  - Writes out a single bit
- writeChar
  - Writes out a single (8 bit) character
- writeInt
  - Writes out a single 9 bit integer. If the value passed in is greater than 255, or less than -255, value printed out is not guaranteed

10-42: Binary Files

- If we write to a binary file:
  - bit, bit, char, bit, int
- And then read from the file:
  - bit, char, bit, int, bit
- What will we get out?

10-43: Binary Files

- If we write to a binary file:
  - bit, bit, char, bit, int
- And then read from the file:
  - bit, char, bit, int, bit
- What will we get out?
- Garbage! (except for the first bit)

10-44: 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)
  - Print out *9 bits* for each node – positive values for leaves, negative values for interior nodes
    - Value printed for interior nodes doesn't matter, as long as it is negative

10-45: Command Line Arguments

```
public static void main(String args[])
```

- The `args` parameter holds the input parameters
- java MyProgram arg1 arg2 arg3
  - args.length() = 3
  - args[0] = “arg1”
  - args[1] = “arg2”
  - args[2] = “arg3”

10-46: Calling Huffman

```
java Huffman (-c|-u) [-v] infile outfile
```

- `(-c|-u)` stands for either “-c” (for compress), or “-u” (for uncompress)
- `[-v]` stands for an optional “-v” flag (for verbose)
- `infile` is the input file
- `outfile` is the output file