CSE 143
Lecture 18

Huffman

slides created by Ethan Apter
http://www.cs.washington.edu/143/

Huffman Tree

- For your next assignment, you'll create a "Huffman tree"
- Huffman trees are using for file compression
- **file compression:** making files smaller
  - for example, WinZip makes zip files
- Huffman trees allow us to implement a relatively simple form of file compression
  - Huffman trees are essentially just binary trees
  - it's not as good as WinZip, but it's a whole lot easier!
- Specifically, we're going to compress text files

ASCII

- Characters in a text file are all encoded by bits
  - **bit**: the smallest piece of information on a computer ("zero" or "one")
  - your computer automatically converts the bits into the characters you expect to see
- Normally, all characters are encoded by the same number of bits
  - this makes it easy to find the boundaries between characters
- One character encoding is the American Standard Code for Information Interchange
  - better known as ASCII

ASCII

- The original version of ASCII had 128 characters
  - this fit perfectly into 7 bits ($2^7 = 128$)
- But the standard data size is 8 bits (i.e. a byte)
  - original ASCII used the 8^{th} bit as a "parity" (odd or even) bit
  - ...which didn't work out very well
- Eventually, 128 characters wasn't enough
- Extended ASCII has 256 characters
  - this fits perfectly into 8 bits ($2^8 = 256$)
  - can represent 00000000 to 11111111 (binary)
  - can represent 0 to 255 (decimal)

Text Files

- In simple text files, each byte (8 bits) represents a single character

- If we want to compress the file, we have to do better
  - otherwise, we won't improve the old file

- What if different characters are represented by different numbers of bits?
  - characters that appear frequently will require fewer bits
  - characters that appear infrequently will require more bits

- The Huffman algorithm finds an ideal variable-length way of encoding the characters for a specific file

Huffman Algorithm

- The Huffman algorithm creates a Huffman tree
- This tree represents the variable-length character encoding
- In a Huffman tree, the left and right children each represent a single bit of information
  - going left is a bit of value zero
  - going right is a bit of value one
- But how do we create the Huffman tree?

Creating a Huffman Tree

- First, we have to know how frequently each character occurs in the file

- Then, we construct a leaf node for every character that occurs at least once (i.e. has non-zero frequency)
  - we don't care about letters that never occur, because we won't have to encode them in this particular file

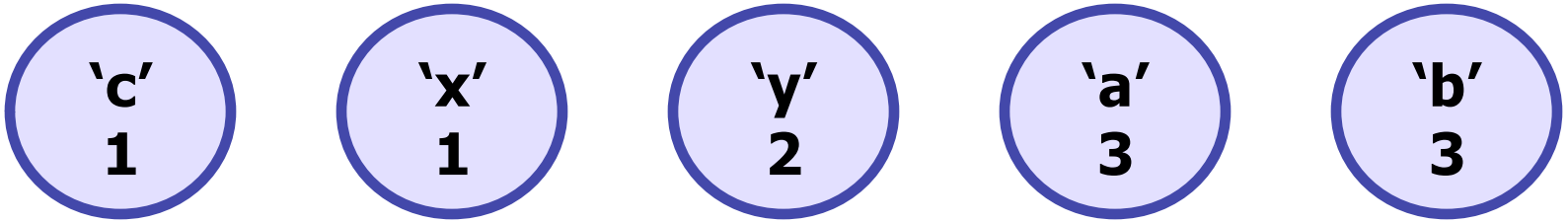
- We now have a list of nodes, each of which contains a character and a frequency

Creating a Huffman Tree

- So we've got a list of nodes
  - we can also think of these nodes as subtrees
- Until we're left with a single tree
  - pick the two subtrees with the smallest frequencies
  - combine these nodes into a new subtree
    - this subtree has a frequency equal to the sum of the two frequencies of its children
- Now we've got our Huffman Tree

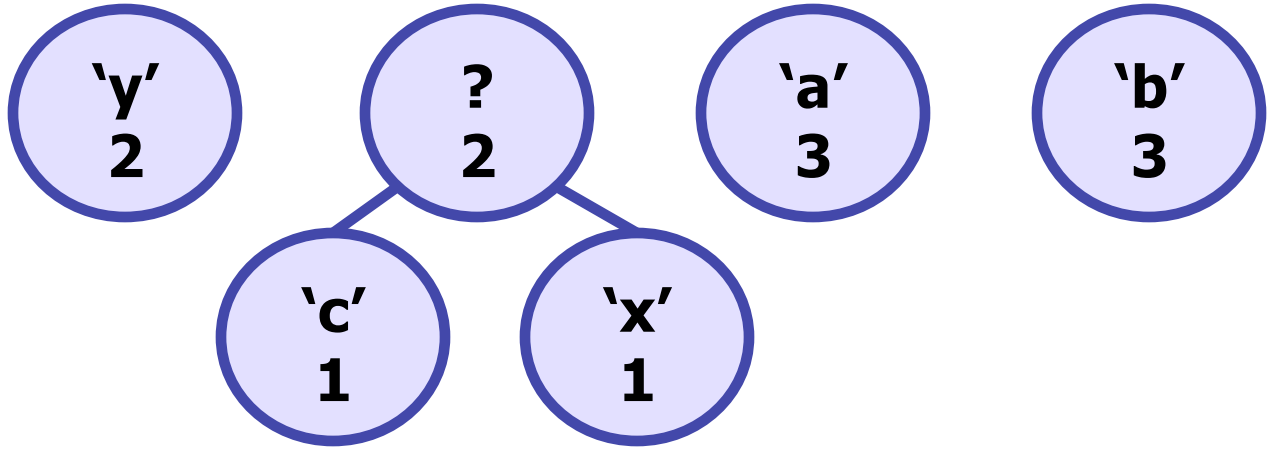
Creating a Huffman Tree

- Visual example of the last few slides:
  - Suppose the file has 3 'a's, 3 'b's, 1 'c', 1 'x', and 2 'y's
  - (subtrees displayed in sorted order, according to frequency)



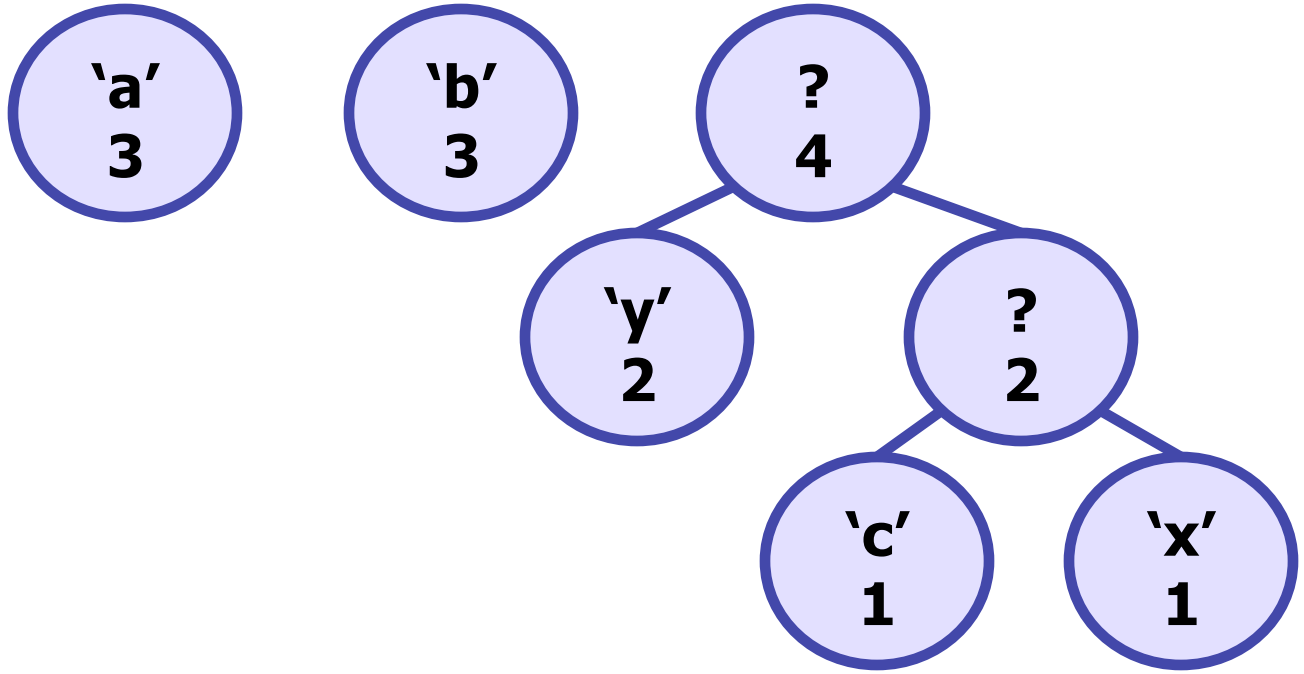
Creating a Huffman Tree

- Visual example of the last few slides:
  - Suppose the file has 3 'a's, 3 'b's, 1 'c', 1 'x', and 2 'y's
  - (subtrees displayed in sorted order, according to frequency)



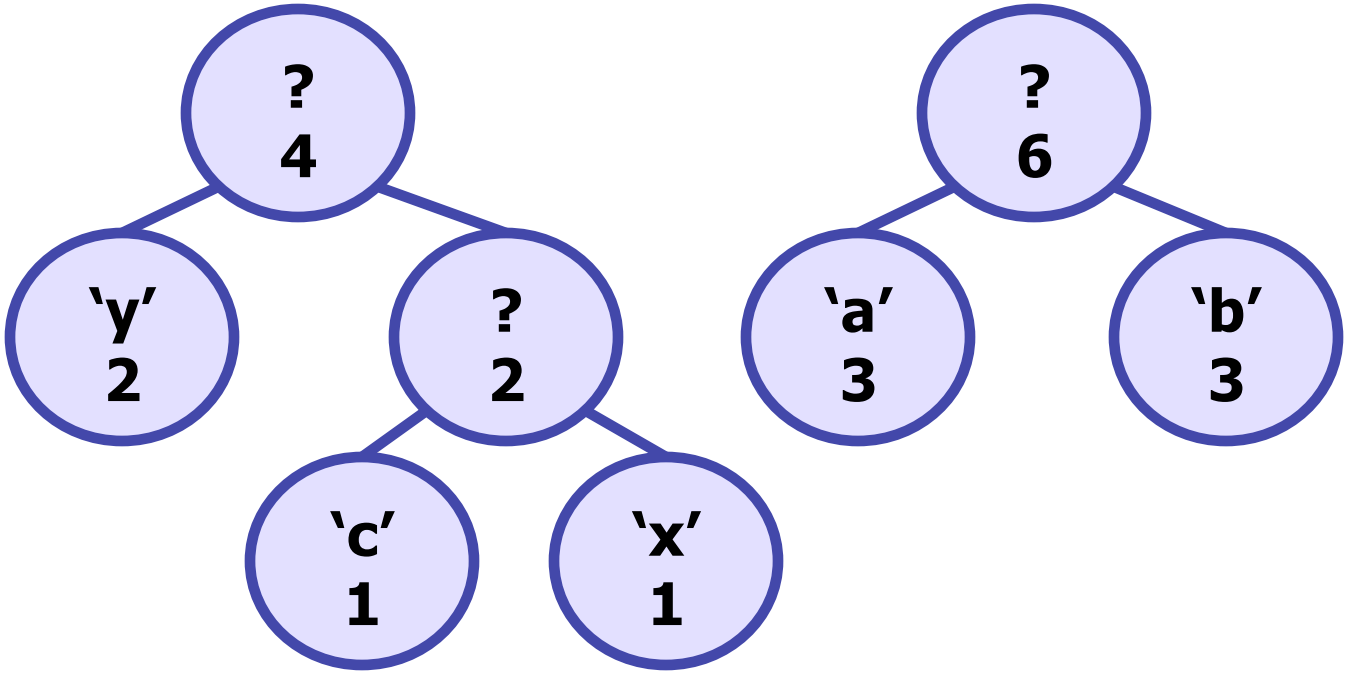
Creating a Huffman Tree

- Visual example of the last few slides:
  - Suppose the file has 3 'a's, 3 'b's, 1 'c', 1 'x', and 2 'y's
  - (subtrees displayed in sorted order, according to frequency)



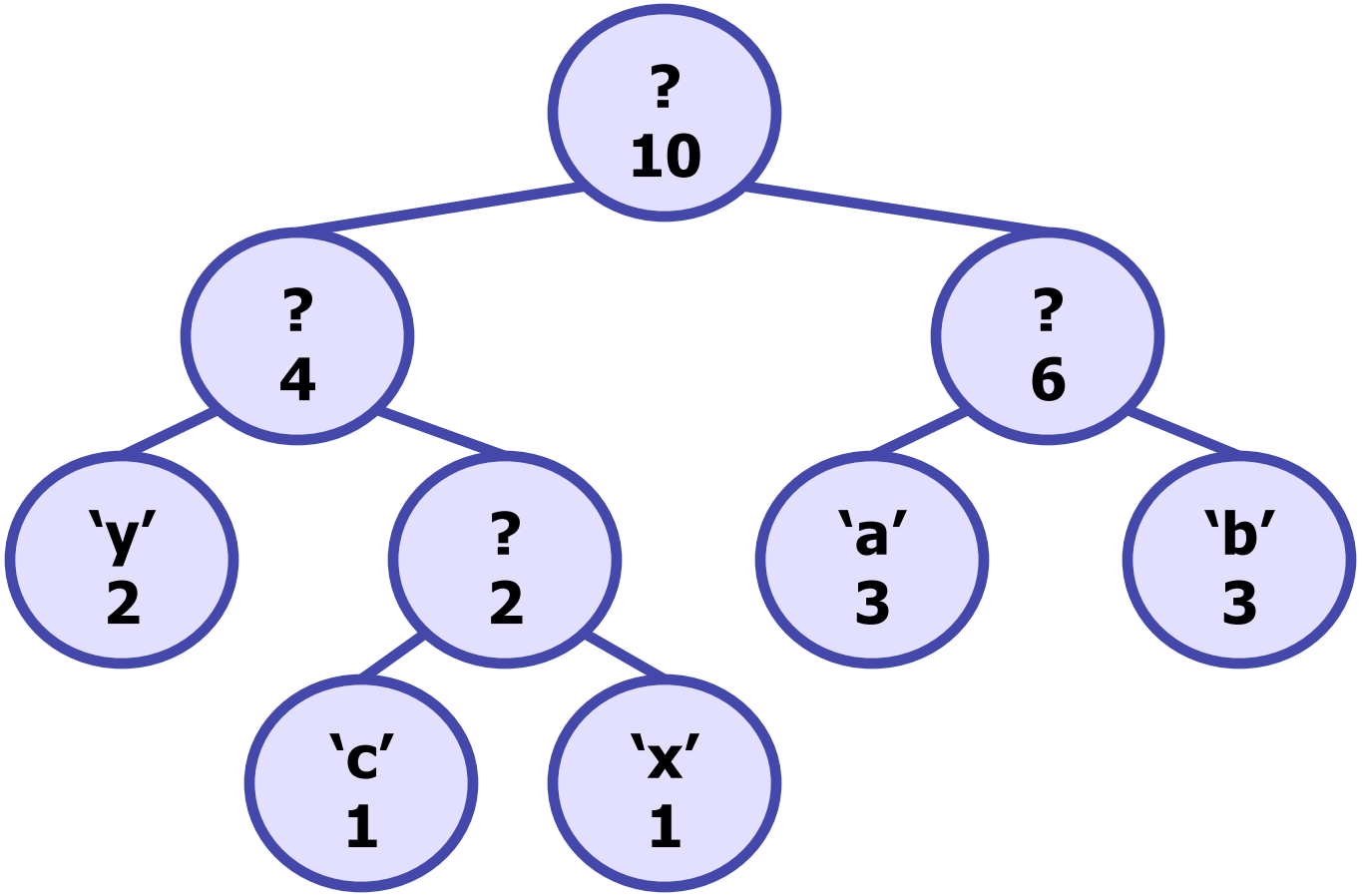
Creating a Huffman Tree

- Visual example of the last few slides:
  - Suppose the file has 3 'a's, 3 'b's, 1 'c', 1 'x', and 2 'y's
  - (subtrees displayed in sorted order, according to frequency)



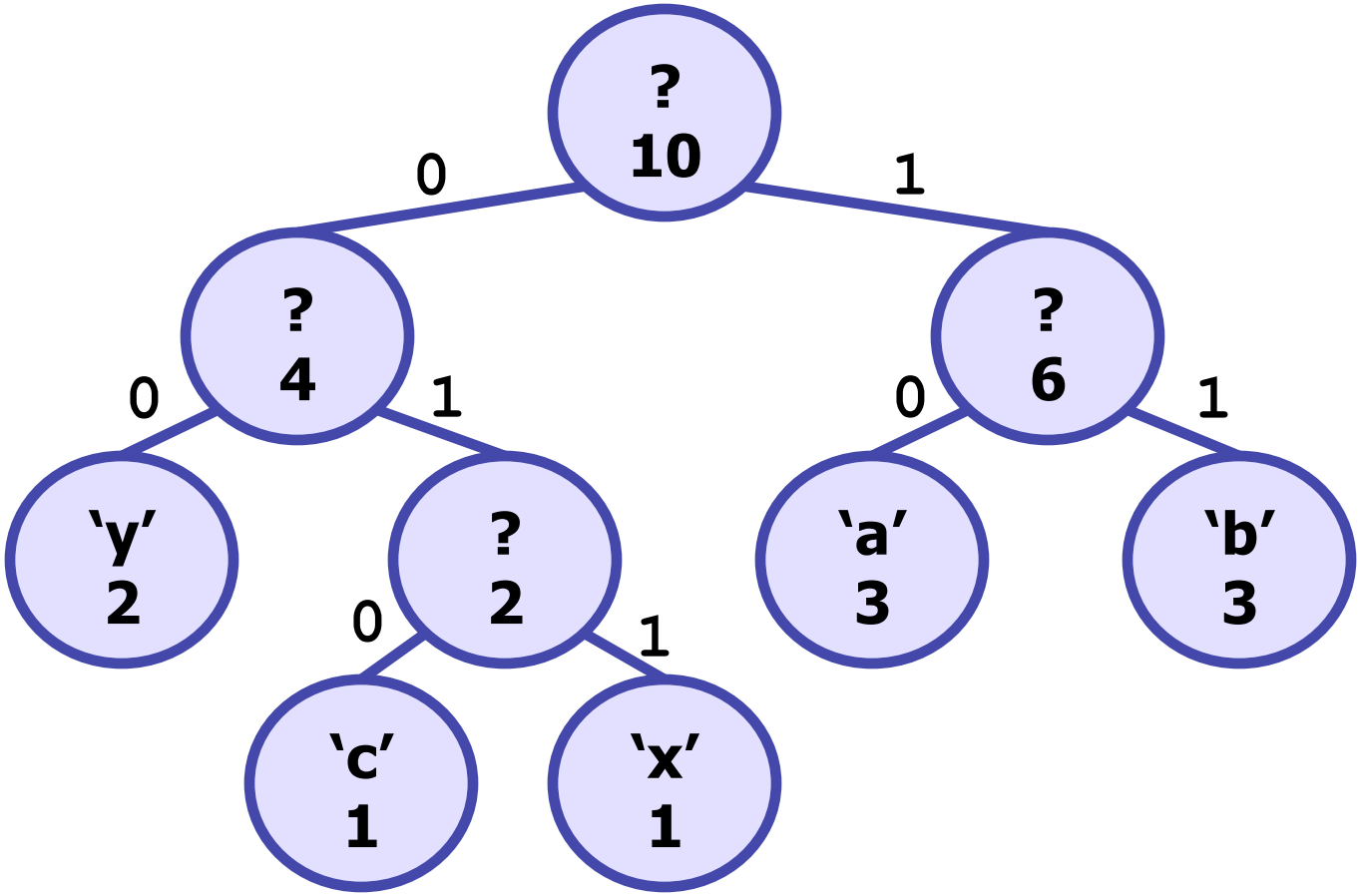
Creating a Huffman Tree

- Visual example of the last few slides:
  - Suppose the file has 3 'a's, 3 'b's, 1 'c', 1 'x', and 2 'y's
  - (subtrees displayed in sorted order, according to frequency)



Creating a Huffman Tree

- Recall also that:
  - moving to the left child means 0
  - moving to the right child means 1



Creating a Huffman Tree

- These are the character encodings for the previous tree:
  - 00 is the character encoding for ‘y’
  - 010 is the character encoding for ‘c’
  - 011 is the character encoding for ‘x’
  - 10 is the character encoding for ‘a’
  - 11 is the character encoding for ‘b’

- Notice that characters with higher frequencies have shorter encodings
  - ‘a’, ‘b’, and ‘y’ all have 2 character encodings
  - ‘c’ and ‘x’ have 3 character encodings

- Once we have our tree, the frequencies don’t matter
  - we just needed the frequencies to compute the encodings

Reading and Writing Bits

- So, the character encoding for ‘x’ is 011

- But we don’t want to write the **`String`** “011” to a file

```
// assume output writes to a file
output.print(“011”); // bad!
```

- Why?
  - we just replaced a single character (‘x’) with three characters (‘0’, ‘1’, and ‘1’)
  - so now we’re using 24 bits instead of just 8 bits!

- Instead, we need a way to read and write a single bit

Reading and Writing Bits

- To write a single bit, Stuart wrote **`BitOutputStream`**
  - The Encode.java program uses **`BitOutputStream`** and the character encodings from your Huffman tree to encode a file

- To read a single bit, Stuart wrote **`BitInputStream`**
  - The Decode.java program opens a **`BitInputStream`** to read the individual bits of the encoded file
  - ...but it passes this **`BitInputStream`** to you and makes you do all the work

- The only method you care about is in **`BitInputStream`**:

```
// reads and returns the next bit in this input stream
public int readBit()
```

Decoding an Encoded File

- To decode a file:
  - Start at the top of the Huffman tree
  - Until you're at a leaf node
    - Read a single bit (0 or 1)
    - Move to the appropriate child (0 → left, 1 → right)
  - Write the character at the leaf node
  - Go back to the top of the tree and repeat until you've decoded the entire file

End of File

- But how do we know when the file ends?

- Every file must consist of a whole number of bytes
  - so the number of bits in a file must be a multiple of 8

- This was fine when every character was also exactly one byte, but it might not work out well with our variable-length encodings
  - Suppose your encoding of a file is 8001 bits long
  - Then the resulting encoded file will have 8008 bits
  - Clearly, there are 7 bits at the end of the encoded file that don't correspond data in the original file
  - But 7 is a lot of bits for Huffman, and it's likely that we would decode a few extra characters

End of File

- To get around this, we're going to introduce a "fake" character at the end of our file
  - we'll call this fake character the "pseudo-eof" character
  - "pseudo-eof": pseudo end-of-file

- This character does not actually exist in the original file
  - it just lets us know when to stop

- Because the pseudo-eof is fake, it should have a character value different than the other characters
  - characters have values 0 to 255, so our pseudo-eof will have value 256 (i.e. one larger than the largest character value)

End of File

- Using the pseudo-eof when creating a Huffman tree:
  - you'll need to create a leaf node containing the character value for your pseudo-eof with frequency 1 when you're creating the other leaf nodes
  - the rest of the algorithm stays the same
- Using the pseudo-eof when decoding a file:
  - if you ever decode the pseudo-eof (i.e. reach the leaf node containing the pseudo-eof), you need to stop decoding
  - we don't want to decode the value of the pseudo-eof because the pseudo-eof is completely fake

Using `HuffmanTree`

- There are three main/client programs for this assigment
- MakeCode.java outputs the character encoding to a file
  - you must complete the first part of the assignment to use MakeCode.java
- Encode.java takes a text file and a character encoding file. It uses these files to output an encoded file.
- Decode.java takes an encoded file and a character encoding file. It uses these files to output a decoded file.
  - you must complete the second part of the assignment to use Decode.java

Using `HuffmanTree`

- Using the three main/client programs on hamlet.txt
- We give hamlet.txt to MakeCode.java. MakeCode.java produces the character encoding file (which we'll call hamlet.code)
- We give hamlet.txt and hamlet.code to Encode.java. Encode.java produces the encoded file (which we'll call hamlet.short)
- We give hamlet.short and hamlet.code to Decode.java. Decode.java produces a decoded file (which we'll call hamlet.new). hamlet.new is identical to hamlet.txt.