CSE 143
Lecture 22

Huffman

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

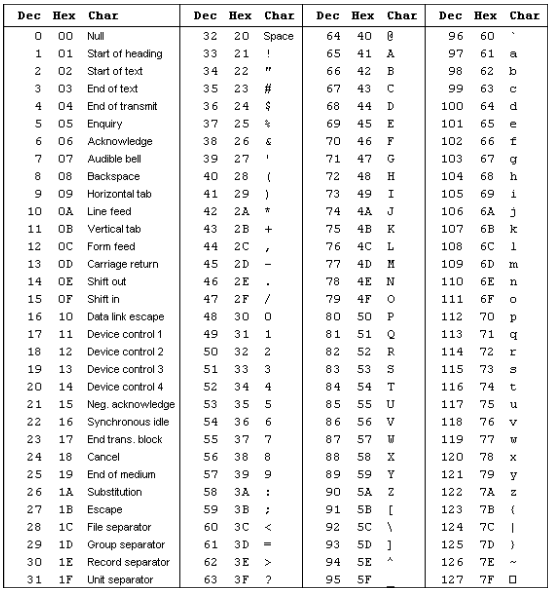
Huffman Tree

- For your next assignment, you'll create a "Huffman tree"
- Huffman trees are used 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 Table

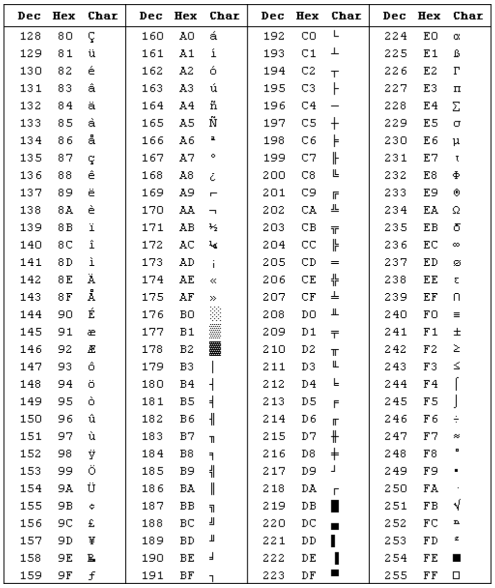


Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	Null	32	20	Space	64	40	@	96	60	`
1	01	Start of heading	33	21	!	65	41	A	97	61	a
2	02	Start of text	34	22	"	66	42	B	98	62	b
3	03	End of text	35	23	#	67	43	C	99	63	c
4	04	End of transmit	36	24	$	68	44	D	100	64	d
5	05	Enquiry	37	25	%	69	45	E	101	65	e
6	06	Acknowledge	38	26	&	70	46	F	102	66	f
7	07	Audible bell	39	27	'	71	47	G	103	67	g
8	08	Backspace	40	28	(	72	48	H	104	68	h
9	09	Horizontal tab	41	29	)	73	49	I	105	69	i
10	0A	Line feed	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage return	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	47	2F	/	79	4F	O	111	6F	o
16	10	Data link escape	48	30	0	80	50	P	112	70	p
17	11	Device control 1	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	50	32	2	82	52	R	114	72	r
19	13	Device control 3	51	33	3	83	53	S	115	73	s
20	14	Device control 4	52	34	4	84	54	T	116	74	t
21	15	Neg. acknowledge	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	54	36	6	86	56	V	118	76	v
23	17	End trans. block	55	37	7	87	57	W	119	77	w
24	18	Cancel	56	38	8	88	58	X	120	78	x
25	19	End of medium	57	39	9	89	59	Y	121	79	y
26	1A	Substitution	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	63	3F	?	95	5F	_	127	7F	□

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 8th 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)

Extended ASCII Table



Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80	Ç	160	A0	á	192	C0	└	224	E0	α
129	81	ü	161	A1	í	193	C1	┴	225	E1	ß
130	82	é	162	A2	ó	194	C2	┬	226	E2	Γ
131	83	â	163	A3	ú	195	C3	├	227	E3	π
132	84	ä	164	A4	ñ	196	C4	─	228	E4	Σ
133	85	à	165	A5	Ñ	197	C5	┼	229	E5	σ
134	86	å	166	A6	ª	198	C6	╞	230	E6	µ
135	87	ç	167	A7	º	199	C7	╟	231	E7	τ
136	88	ê	168	A8	¿	200	C8	╚	232	E8	Φ
137	89	ë	169	A9	⌐	201	C9	╔	233	E9	Θ
138	8A	è	170	AA	¬	202	CA	╩	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	╦	235	EB	δ
140	8C	î	172	AC	¼	204	CC	╠	236	EC	∞
141	8D	ì	173	AD	¡	205	CD	═	237	ED	φ
142	8E	Ä	174	AE	«	206	CE	╬	238	EE	ε
143	8F	Å	175	AF	»	207	CF	╧	239	EF	∩
144	90	É	176	B0	░	208	D0	╨	240	F0	≡
145	91	æ	177	B1	▒	209	D1	╤	241	F1	±
146	92	Æ	178	B2	▓	210	D2	╥	242	F2	≥
147	93	ô	179	B3	│	211	D3	╙	243	F3	≤
148	94	ö	180	B4	┤	212	D4	╘	244	F4	⌠
149	95	ò	181	B5	╡	213	D5	╒	245	F5	⌡
150	96	û	182	B6	╢	214	D6	╓	246	F6	÷
151	97	ù	183	B7	╖	215	D7	╫	247	F7	≈
152	98	ÿ	184	B8	╕	216	D8	╪	248	F8	°
153	99	Ö	185	B9	╣	217	D9	┘	249	F9	∙
154	9A	Ü	186	BA	║	218	DA	┌	250	FA	·
155	9B	¢	187	BB	╗	219	DB	█	251	FB	√
156	9C	£	188	BC	╝	220	DC	▄	252	FC	ⁿ
157	9D	¥	189	BD	╜	221	DD	▌	253	FD	²
158	9E	₧	190	BE	╛	222	DE	▐	254	FE	■
159	9F	ƒ	191	BF	┐	223	DF	▀	255	FF	□

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

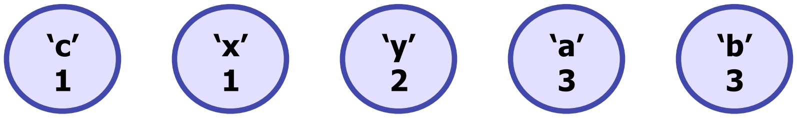
- First, we have to know how frequently each character occurs in the file
- Then, we construct a leaf node for every character that occurs at least once (i.e. has non-zero frequency)
  - we don't care about letters that never occur, because we won't have to encode them in this particular file
- We now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

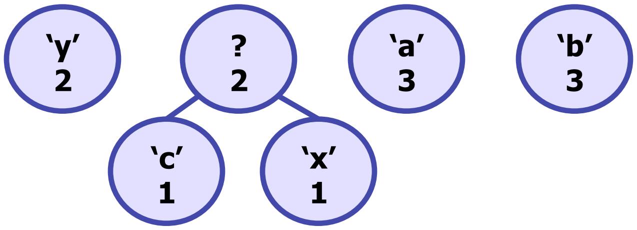
Creating a Huffman Tree

- 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)



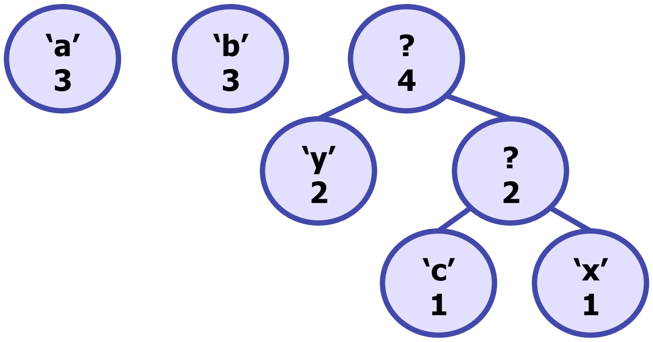
Creating a Huffman Tree

- 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)



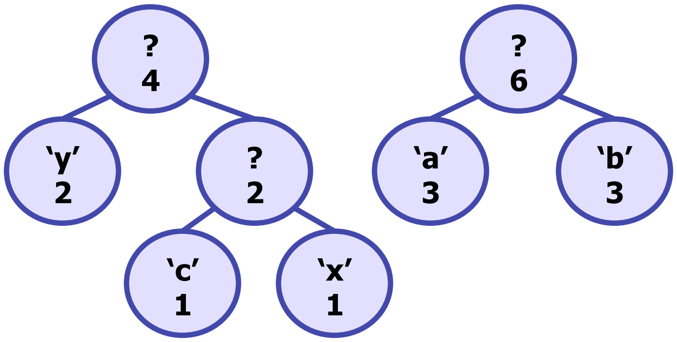
Creating a Huffman Tree

- 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)



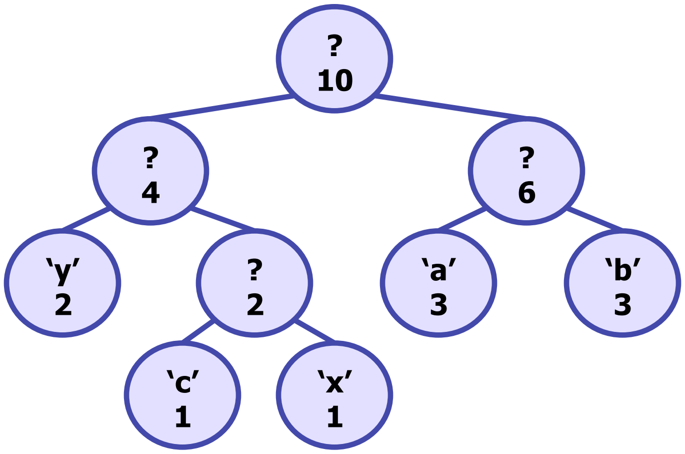
Creating a Huffman Tree

- 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)



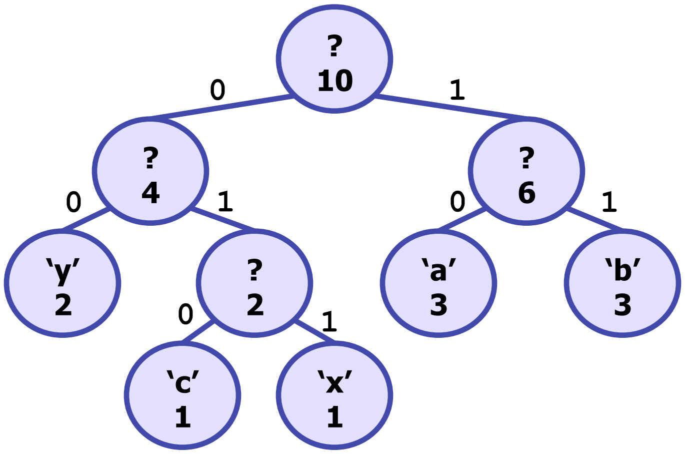
Creating a Huffman Tree

- 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 Reges (author of book) 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()
```

Bit Input/Output Streams

- `BitInputStream`: like any other stream, but allows you to read one bit at a time from input until it is exhausted.

`public BitInputStream(String file)`	Creates stream to read bits from file with given name
`public int readBit()`	Reads a single 1 or 0; returns -1 at end of file
`public void close()`	Stops reading from the stream

- `BitOutputStream`: same, but allows you to *write* one bit at a time.

`public BitOutputStream(String file)`	Creates stream to write bits to file with given name
`public void writeBit(int bit)`	Writes a single bit
`public void close()`	Stops reading from the stream

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 X, so our pseudo-eof will have value (X+1) (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)

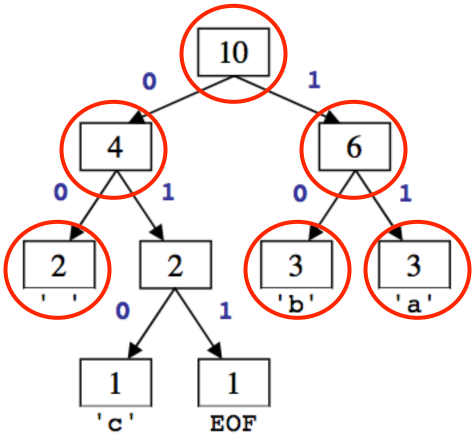
- We give hamlet.short and hamlet.code to Decode.java. Decode.java produces a decoded file (which we'll call hamlet.new). hamlet.new is identical to hamlet.txt.

HuffmanTree: Part II

- Given a bunch of bits, how do we decompress them?
- Hint: HuffmanTrees have an encoding "prefix property."
  - No encoding A is the prefix of another encoding B
  - I.e. never will *x* → `011` and *y* → `011100110` be true for any two characters *x* and *y*
- Tree structure tells how many bits represent "next" character
- While there are more bits in the input stream:
  - Read a bit
  - If zero, go left in the tree; if one, go right
  - If at a leaf node, output the character at that leaf and go back to the tree root

HuffmanTree: Part II cont'd.

HuffmanTree for `"ab ab cab"`



Sample encoding

`111000...`

→ `"ab  "`