Priority Queues Part #2

Re-submit Assignment

Due Jun 6, 2017 by 11:59pm **Points** 100 **Submitting** a file upload **File Types** zip

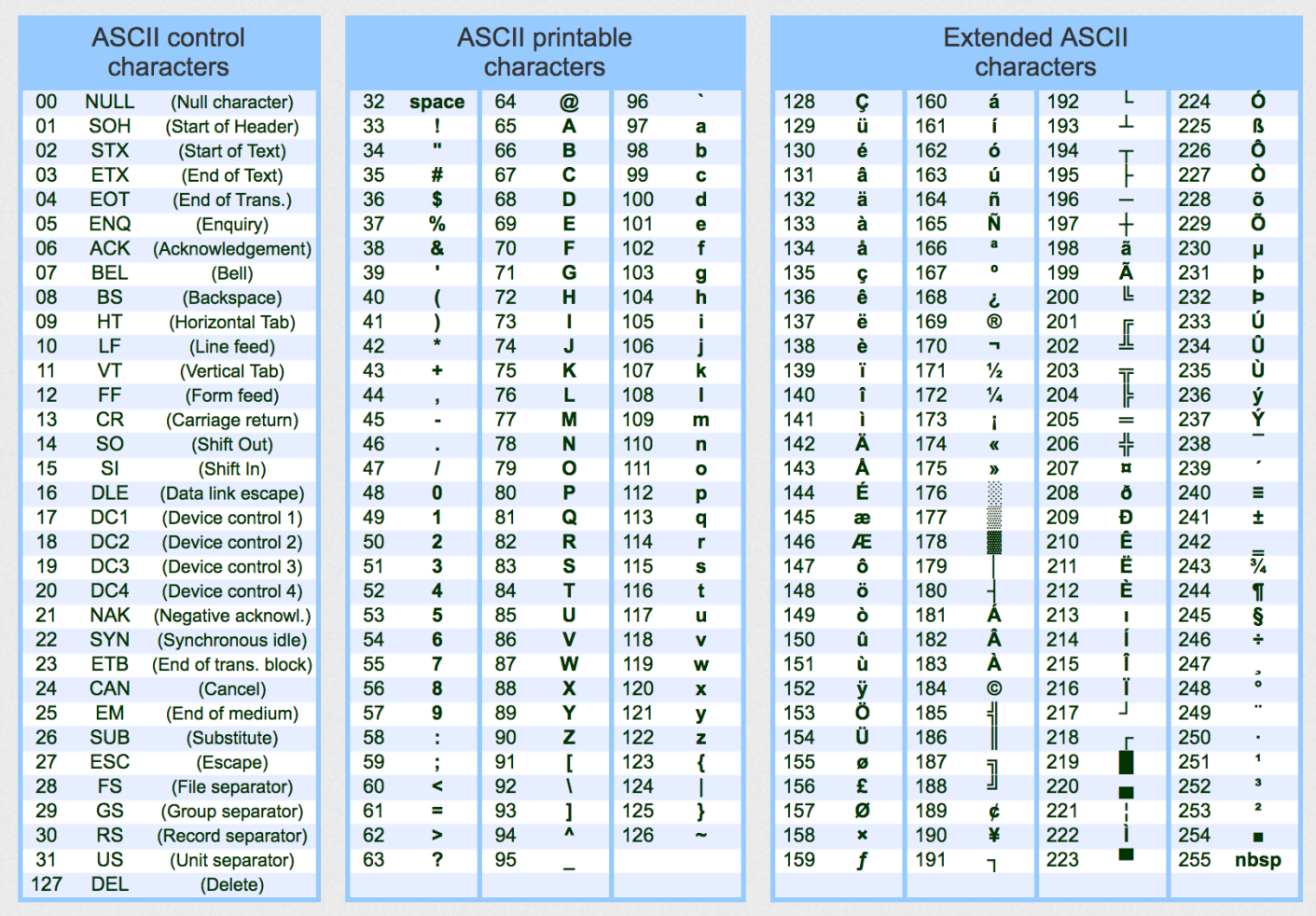
Overview

This assignment will focus on an application of priority queues called Huffman Encoding. This encoding allows a developer to compress data stored in text files so that they use up less space on disk.

Character Encoding

Every character in a program that is ran on a computer is stored as a (binary) number, using an encoding scheme. You have probably heard of several of these schemes before: ASCII, Unicode, UTF-8, UTF-16, etc. Each scheme creates a mapping between textual characters and a number (in binary) that will be stored on disk. Encoding schemes have a difficult job to accomplish as there are many different languages and character sets used in the world, yet the scheme must try to store characters using as few binary bits as possible.

Below shows the simplest of schemes, an ASCII table:



ASCII control characters

00	NULL	(Null character)
01	SOH	(Start of Header)
02	STX	(Start of Text)
03	ETX	(End of Text)
04	EOT	(End of Trans.)
05	ENQ	(Enquiry)
06	ACK	(Acknowledgement)
07	BEL	(Bell)
08	BS	(Backspace)
09	HT	(Horizontal Tab)
10	LF	(Line feed)
11	VT	(Vertical Tab)
12	FF	(Form feed)
13	CR	(Carriage return)
14	SO	(Shift Out)
15	SI	(Shift In)
16	DLE	(Data link escape)
17	DC1	(Device control 1)
18	DC2	(Device control 2)
19	DC3	(Device control 3)
20	DC4	(Device control 4)
21	NAK	(Negative acknowl.)
22	SYN	(Synchronous idle)
23	ETB	(End of trans. block)
24	CAN	(Cancel)
25	EM	(End of medium)
26	SUB	(Substitute)
27	ESC	(Escape)
28	FS	(File separator)
29	GS	(Group separator)
30	RS	(Record separator)
31	US	(Unit separator)
127	DEL	(Delete)

ASCII printable characters

32	space	64	@	96	`
33	!	65	A	97	a
34	"	66	B	98	b
35	#	67	C	99	c
36	$	68	D	100	d
37	%	69	E	101	e
38	&	70	F	102	f
39	'	71	G	103	g
40	(	72	H	104	h
41	)	73	I	105	i
42	*	74	J	106	j
43	+	75	K	107	k
44	,	76	L	108	l
45	-	77	M	109	m
46	.	78	N	110	n
47	/	79	O	111	o
48	0	80	P	112	p
49	1	81	Q	113	q
50	2	82	R	114	r
51	3	83	S	115	s
52	4	84	T	116	t
53	5	85	U	117	u
54	6	86	V	118	v
55	7	87	W	119	w
56	8	88	X	120	x
57	9	89	Y	121	y
58	:	90	Z	122	z
59	;	91	[	123	{
60	<	92	\	124	|
61	=	93	]	125	}
62	>	94	^	126	~
63	?	95	_		

Extended ASCII characters

128	Ç	160	á	192	└	224	Ó
129	ü	161	í	193	┴	225	ß
130	é	162	ó	194	┬	226	Ô
131	â	163	ú	195	├	227	Ò
132	ä	164	ñ	196	─	228	õ
133	à	165	Ñ	197	┼	229	Õ
134	å	166	ª	198	ã	230	µ
135	ç	167	º	199	Ã	231	þ
136	ê	168	¿	200	╚	232	Þ
137	ë	169	®	201	╔	233	Ú
138	è	170	¬	202	╩	234	Û
139	ï	171	½	203	╦	235	Ù
140	î	172	¼	204	╠	236	ý
141	ì	173	¡	205	═	237	Ý
142	Ä	174	«	206	╬	238	¯
143	Å	175	»	207	¤	239	´
144	É	176	░	208	ð	240	≡
145	æ	177	▒	209	Ð	241	±
146	Æ	178	▓	210	Ê	242	‗
147	ô	179	│	211	Ë	243	¾
148	ö	180	┤	212	È	244	¶
149	ò	181	Á	213	ı	245	§
150	û	182	Â	214	Í	246	÷
151	ù	183	À	215	Î	247	¸
152	ÿ	184	©	216	Ï	248	°
153	Ö	185	╣	217	┘	249	¨
154	Ü	186	║	218	┌	250	·
155	ø	187	╗	219	█	251	¹
156	£	188	╝	220	▄	252	³
157	Ø	189	¢	221	¦	253	²
158	×	190	¥	222	Ì	254	■
159	ƒ	191	┐	223	▀	255	nbsp

Notice how each character is mapped to a numeric value. This value is then stored in binary on disk representing the associated character.

Huffman Encoding

The Huffman encoding scheme attempts to use the structure of a text document to reduce the amount of binary data stored on disk. The key observation the algorithm makes is that certain characters appear more often in text, while other characters appear less often. For example you can make the following observations easily:

In an English sentence:

- The characters e, r, s, and t typically show up more often than other characters
- The characters q, x, y and z typically show up less often than other characters

The Huffman algorithm uses less bits to store frequently used characters, while less frequently used characters may take up more bits. To accomplish this the algorithm first needs a frequency chart of the characters in a document. Below is such a chart for the book "War and Peace" (1200+ pages)

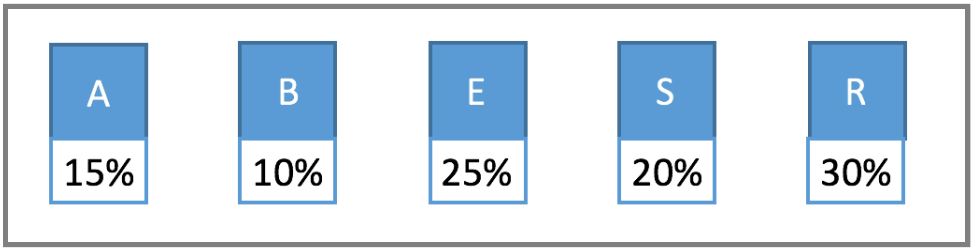
```
 : 0.17016000942587098
a: 0.06671732852912311
b: 0.011406488711663791
c: 0.020280761942124362
```

```
d: 0.0389337180914191
e: 0.10320242650354819
f: 0.018068775946651676
g: 0.016892516766794603
h: 0.055098889366472205
i: 0.05669246713616105
j: 0.00084714357273422
k: 0.006724490084734104
l: 0.03177018778678311
m: 0.02028964806351668
n: 0.060617828982315304
o: 0.062559281948733
p: 0.014985620939124802
q: 0.0007671684802033671
r: 0.048850958680851986
s: 0.05361194505349116
t: 0.07451638107111333
u: 0.02119471598310452
v: 0.008914754450136682
w: 0.01948660598213692
x: 0.0014428428216265814
y: 0.015216660095325045
z: 0.0007.5038358424010
```

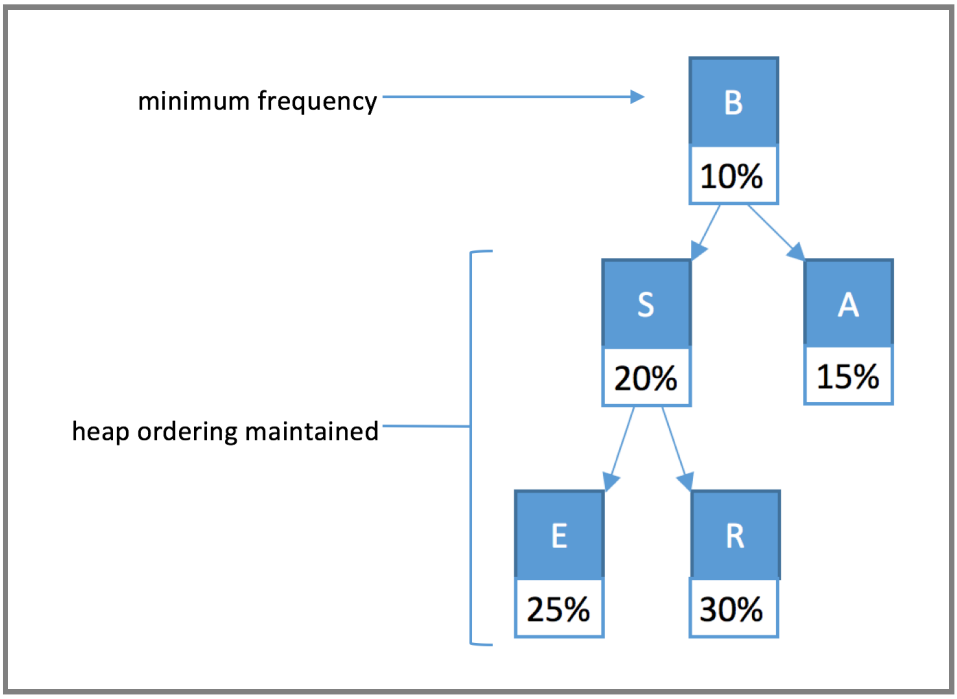
Please note the following:

- Each of the values above represents the percentage of times the character was encountered. For example, "e" was encountered roughly 10% of the time, whereas "r" as encountered 4% of the time.
- The output above first converted the entire book to lowercase letters. So "a" is considered the same as "A." This is just a simplification of the problem to smooth the difficulty of this assignment.
- The first value in the table is the space " " character.

Each character is then placed in a binary tree node, where the node stores both the character and frequency of that character appearing in the document:

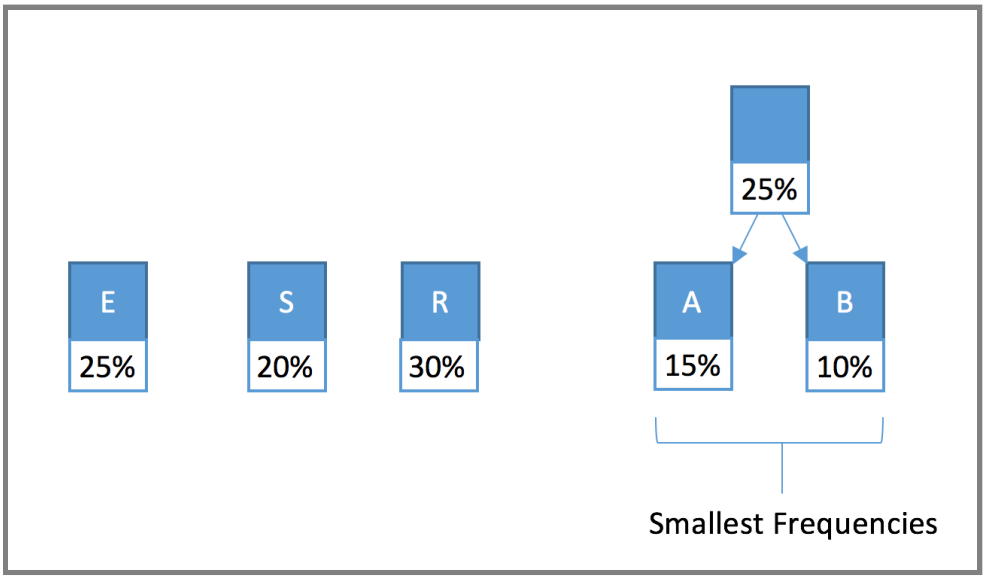


These tree nodes are then placed in a min-heap ordered by the frequency of each node:



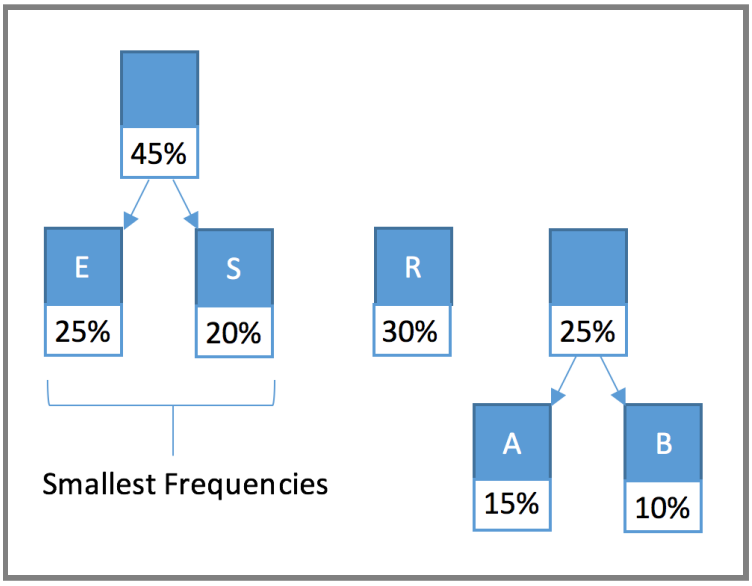
The algorithm then performs the following steps repeatedly until the heap has only one element:

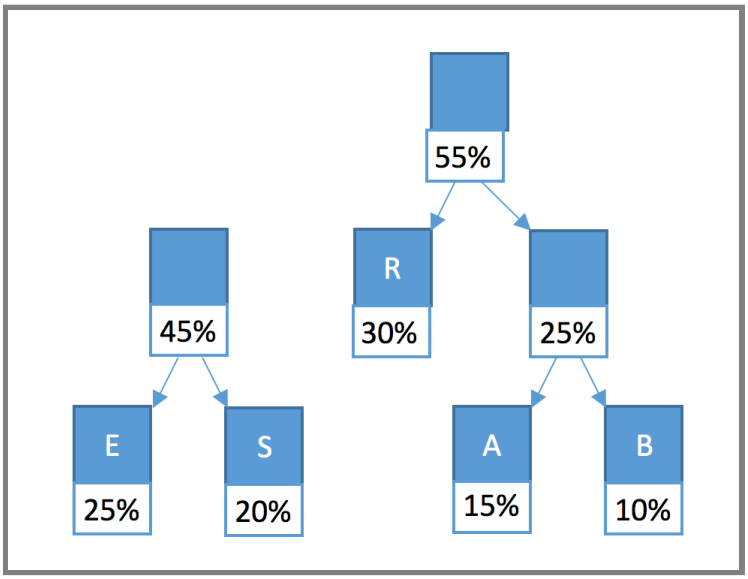
- Remove the two lowest frequency nodes from the heap
- Create a new parent node for the two nodes removed from the heap.
  - The parent node stores no character. You can represent this by storing the null character: '\u0000'
  - The parent node's frequency is the sum of both of the child node frequencies



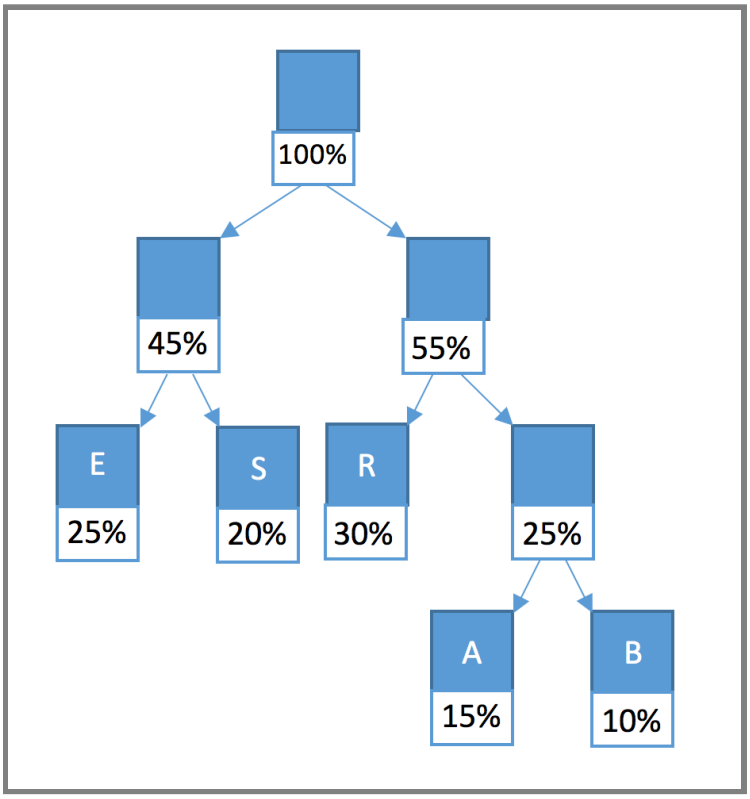
- The new parent node is then put back on the heap

Here are a few steps of the algorithm to show how this might occur:



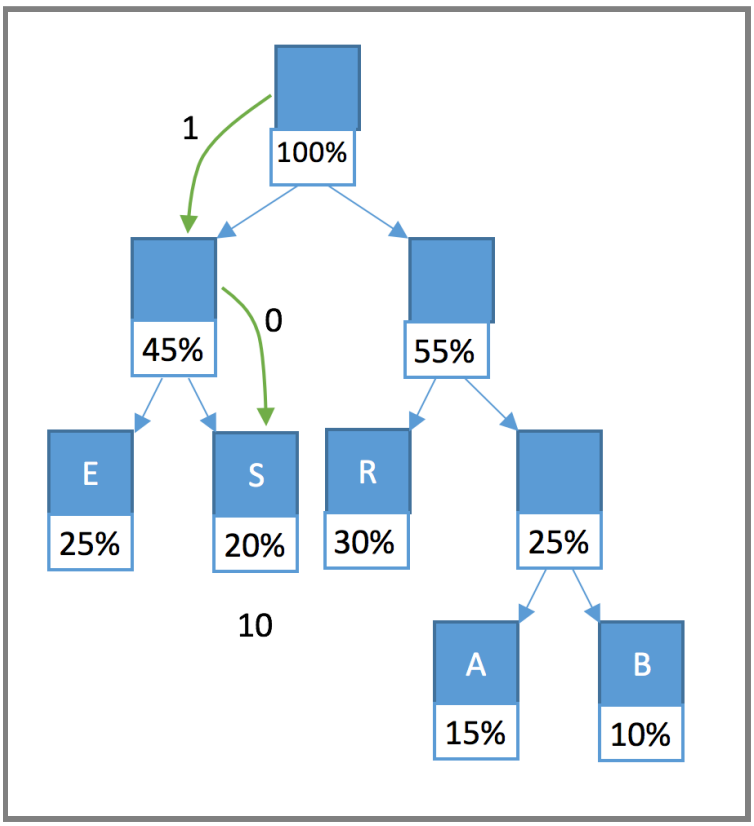


After all nodes have been assembled into a single tree, the frequency of the remaining node (the root of the tree) will be roughly 1.0 (or 100%).

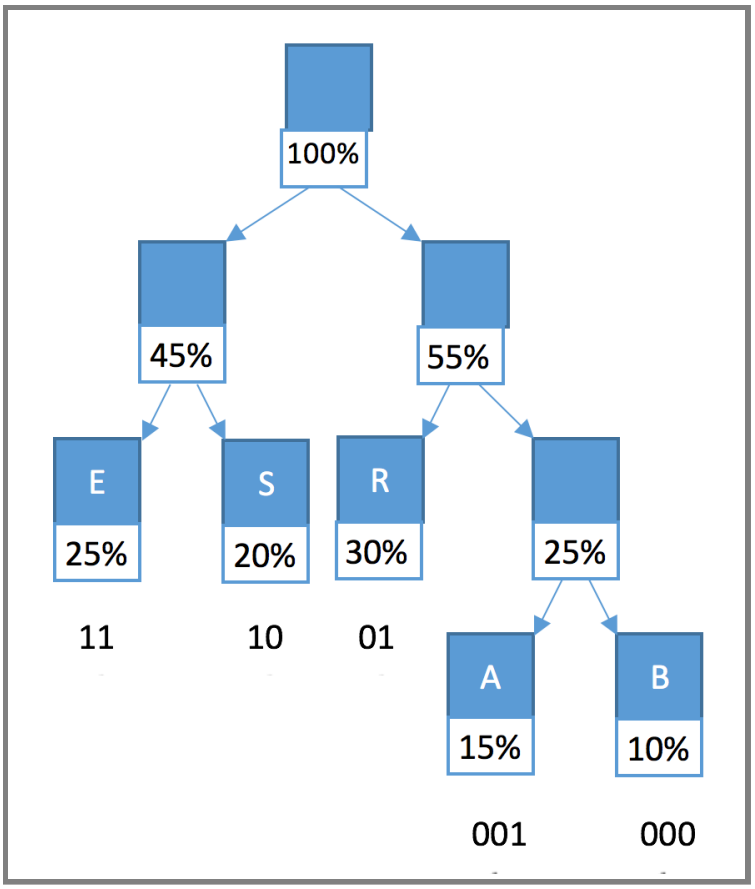


Generating Huffman Values

The tree created by the Huffman algorithm can be used to generate binary strings with a shorter length for higher frequency characters and a longer length for lower frequency characters. To convert a character to it's binary equivalent, you must traverse the tree for each character stored in the tree. As you traverse the tree, you will build a binary string based on whether you move to a left child (a binary 1) or a right child (a binary 0). The path you take to each node containing a character determines it's final binary value, as seen below:



Here is the entire set of Huffman values for the small tree generated above. Notice how there are fewer characters than ASCII for example (which usually uses 7-8 bits).



Below can be seen the Huffman values generated from the frequencies above. This may seem like small savings (in bits used), but keep in mind that the shorter binary strings " " (space), "e", "o", "r", "s", "t", are used more often in the document, so the savings will be compounded.

```
 : 000
a: 0101
b: 100011
c: 001000
d: 00101
e: 110
f: 010001
g: 010010
h: 1010
i: 1001
j: 0100000101
k: 01000000
l: 10000
m: 11111
n: 0111
o: 0110
p: 100010
```

```
q: 0100000110
r: 1110
s: 1011
t: 0011
u: 11110
v: 0100001
w: 001001
x: 0100000100
y: 010011
z: 0100000111
```

Huffman Encoding Program

Your task for this assignment is to generate a Huffman encoding tree for the book "War and Peace" and to visually show (on the console) the compression of bits from the algorithm. To start download the following text files, found **here**:

- war_and_peace.txt: the full text of "War and Peace"
- war_and_peace_(short).txt: an excerpt from the larger book file that you can use for testing your algorithm
- ascii_to_binary.txt: this file contains a table (mapping) of each character we will track in the program and it's associated ASCII value in binary.

Note: All special characters have been removed from files to simplify your output in the program.

ASCII Binary Values

Your program should first create a method that will do the following:

- Open a file and read out each character one-by-one
- Converts the character from the file to it's binary ASCII value using the contents of the file above
- Prints the binary values to the console
- Prints the total number of bits used by the entire file (the total binary digits)

Running your program against the smaller "War and Peace" file should print out the following:

```
ASCII Output

01100101011011000110110000100000011100100110100101101110011001100011...
01110101011011110110111001100001011100000110000101110010011101000...
01101001011001100010000001111001011011110111010100100000011100011...
01101110011101000110100101100011011010000111001001101001011100011...
01101101011011110111001001100101001000000111010001101111001000000...
```

```
01100110011000010110100101110100011010000110011001110101011011100...
01101000011000010111011001100101001000000110011001110010011010001...

Total length: 3416
```

Character Frequencies

The next step is to implement Huffman's algorithm. This requires knowing the frequency of each character in your input file. Create a function that accepts a file and returns a Map<Character, Double> that holds characters and their frequencies in the file. Test your file by printing out the frequencies of characters for your test files. For example:

```
Frequencies

 : 0.17016000942587098
a: 0.06671732852912311
b: 0.011406488711663791
c: 0.020280761942124362
d: 0.0389337180914191
...
w: 0.01948660598213692
x: 0.0014428428216265814
y: 0.015216660095325045
z: 7.503835842401017E-4

Root frequency:
0.9999999999999999
```

Building the Tree

To build your Huffman tree you will need to implement a binary tree node class that stores a character and frequency:

```
public class HuffmanNode implements Comparable<HuffmanNode>
{
    private char data;
    private double probability;
    private HuffmanNode left;
    private HuffmanNode right;
```

```
    //more stuff here...
}
```

Tree nodes should be sorted based on their probability. The algorithm begins by placing each of the characters (and their frequencies) into HuffmanNode objects. No node should reference another node when the algorithm starts. You should then place all nodes on min-heap.

Note: This assignment is an extension of your work from the previous assignment. You should be using your M-ary heap to sort Huffman Node objects here.

Following the algorithm description above, you should repeatedly take two nodes from the heap and generate a new parent node with the shared frequency of both child nodes. Instead of storing a character in the parent node, store the null character: '\u0000'. This will help you differentiate between nodes with a character and those without. The parent should then be added back to your heap. You will continue to do this until only one element remains in the heap. This last node will be the root of your Huffman tree.

Generating Huffman Values

Once your tree is available you need to traverse to each and every node containing a character using the algorithm described above. This can be done in a short recursive method. The result of your traversal should also be a Map<Character, Double> similar to your ASCII map above.

With this map available to you, display the file contents and bit length of the smaller text file using your Huffman codes rather than ASCII:

```
Huffman Output

0001101010101100000100010010111010001110000101011100011001010 10...
0100001011001011010111000110000000011000100001110100000111111 00...
1000101101101001010101000110000100111000101010101100110000001 00...
1001001110000111010010100000100000001001111000000001011010101 01...
0111000101000000001110011010111011110101110010011000001100101 11...
1011001101000001100101101100100010101100001101001100010001000 11...
0010101100010001000111101100000010001011110010100110001100100 01...

Total length: 1740
```

Note: how you can visually see the compression in bit length when viewing the output in your console.

Showing the Algorithm

Show the results of analyzing both the short and full text for "War and Peace." Your output should include the frequencies and Huffman values for both files. Print the ASCII and Huffman binary values for just the

short file (as seen below). Your output should follow the text below:

```
Analyzing war and peace (short).txt

Frequencies

 : 0.1836734693877551
a: 0.06575963718820861
b: 0.011337868480725623
c: 0.013605442176870748
d: 0.031746031746031744
...
v: 0.009070294784580499
w: 0.02040816326530612
y: 0.034013605442176874

Root frequency: 1.0000000000000002


Huffman Values

 : 11
a: 0110
b: 0010000
c: 011101
...
v: 0010001
w: 001001
y: 01001


ASCII Output

01100101011011000110110000100000011100100110100101101110...
01110101011011110110111001100001011100000110000101110010 0...
01101001011001100010000001111001011011110111010100100000 0...
01101110011101000110100101100011011010000111001001101001 0...
01101101011011110111001001100101001000000111010001101111 0...
01100110011000010110100101110100011010000110011001110101 0...
01101000011000010111011001100101001000000110011001110010 0...

Total length: 3416


Huffman Output
```

```
0001101010101100000100010010111010001110000101011100011000...
0100001011001011010111000110000000011000100001110100000011...
1000101101101001010101000110000100111000101010110011000...
1001001110000111010010100000100000001001111000000000101101...
0111000101000000001110011010111011110101110010011000001110...
1011001101000001100101101100100010101100001101001100010000...
0010101100010001000111101100000010001011110010100110001100...
```

Total length: 1740

Analyzing war and peace.txt

Frequencies

```
 : 0.17016000942587098
a: 0.06671732852912311
b: 0.011406488711663791
c: 0.020280761942124362
...
x: 0.0014428428216265814
y: 0.015216660095325045
z: 7.503835842401017E-4
```

Root frequency: 0.9999999999999999

Huffman Values

```
 : 000
a: 0101
b: 100011
c: 001000
...
x: 0100000100
y: 010011
z: 0100000111
```

ASCII Output

Total length: 23759640

Huffman Output

```
Total length: 12271924
```

Submission

Submit the code to the dropbox on Canvas.

Priority Queues Part #2 Rubric			
Criteria	**Ratings**		**Pts**
Character frequencies correctly read from input files. Frequencies are printed to the console correctly.	10.0 pts Full Marks	0.0 pts No Marks	10.0 pts
HuffmanNode correctly implemented so that is stores a character and probability. HuffmanNodes are sorted based on probability.	10.0 pts Full Marks	0.0 pts No Marks	10.0 pts
Huffman tree correctly assembled according to the algorithm described above.	10.0 pts Full Marks	0.0 pts No Marks	10.0 pts
All Huffman codes are returned as a Map<Character, Double>, after traversing the Huffman tree.	10.0 pts Full Marks	0.0 pts No Marks	10.0 pts
ASCII binary values are correctly printed for the test files above.	5.0 pts Full Marks	0.0 pts No Marks	5.0 pts
Huffman binary values are correctly printed for the test files above.	5.0 pts Full Marks	0.0 pts No Marks	5.0 pts
Statistics are reported for the number of characters saved by using Huffman encoding on the full "war and peace" text file.	15.0 pts Full Marks	0.0 pts No Marks	15.0 pts
All console output directly follows the images above.	10.0 pts Full Marks	0.0 pts No Marks	10.0 pts
Styles: naming conventions, brace placement, spacing, commented code, redundancy, packages and magic numbers.	15.0 pts Full Marks	0.0 pts No Marks	15.0 pts
Formal documentation: full Javadocs & comment block header.	10.0 pts Full Marks	0.0 pts No Marks	10.0 pts
		Total Points: 100.0	