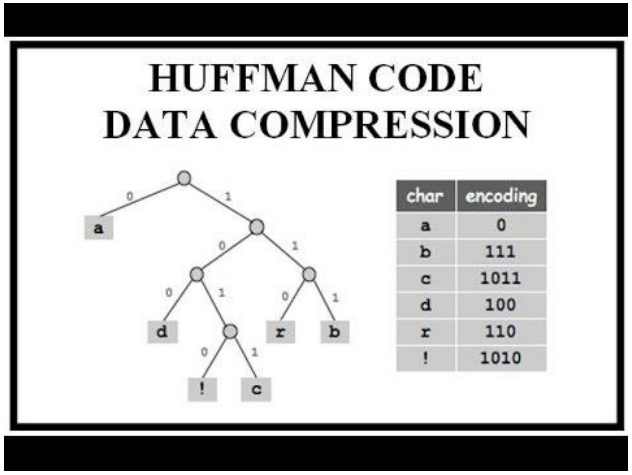
Huffman Code and Data Compression

Textbook Reading:

Chapter 6, Section 6.5, pp. 251-261



Data Compression

- Transporting extremely large files across the internet is commonplace today, and these files must be compressed in order to save time and storage resources.
- For example, there are standard data compression algorithms such as jpeg or gif that are used to compress image files (such as photographs).
- There are also standard data compression algorithms for text and program files, such as those used to zip and unzip files.
- Even before the advent of the Internet, the need for compressing large text files was necessary in order to minimize storage requirements.
- A classical data compression algorithm, due to Huffman in a 1952 paper, is based on a greedy strategy for constructing a code for a given alphabet of symbols. Not only are Huffman codes useful for compressing text files, they also are used as part of image or audio file compression techniques.

Binary Coding

- **Alphabet** of symbols $A = \{a_0, a_1, \ldots, a_{n-1}\}$
- To achieve data compression each symbol in the alphabet is coded with a **binary string**.
- We refer to the set of binary strings encoding the symbols in *A* as a **binary code** for *A*.
- The *ASCII* encoding of the alphabet *A* of standard symbols used by computers consisting of alphabetical characters, numeric characters, punctuation characters, control characters, and so forth. Each symbol of this alphabet is represented by an 8-bit (1-byte) binary string.
- Using a fixed length binary string to store symbols is very convenient for computer implementation, but has the disadvantage that it does not optimize storage usage.
- By allowing **variable length** binary strings for binary codes, we can save space when storing files made up of symbols from our given alphabet.

ASCII TABLE

Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	*[NULL]*
1	1	1	1	*[START OF HEADING]*
2	2	10	2	*[START OF TEXT]*
3	3	11	3	*[END OF TEXT]*
4	4	100	4	*[END OF TRANSMISSION]*
5	5	101	5	*[ENQUIRY]*
6	6	110	6	*[ACKNOWLEDGE]*
7	7	111	7	*[BELL]*
8	8	1000	10	*[BACKSPACE]*
9	9	1001	11	*[HORIZONTAL TAB]*
10	A	1010	12	*[LINE FEED]*
11	B	1011	13	*[VERTICAL TAB]*
12	C	1100	14	*[FORM FEED]*
13	D	1101	15	*[CARRIAGE RETURN]*
14	E	1110	16	*[SHIFT OUT]*
15	F	1111	17	*[SHIFT IN]*
16	10	10000	20	*[DATA LINK ESCAPE]*
17	11	10001	21	*[DEVICE CONTROL 1]*
18	12	10010	22	*[DEVICE CONTROL 2]*
19	13	10011	23	*[DEVICE CONTROL 3]*
20	14	10100	24	*[DEVICE CONTROL 4]*
21	15	10101	25	*[NEGATIVE ACKNOWLEDGE]*
22	16	10110	26	*[SYNCHRONOUS IDLE]*
23	17	10111	27	*[ENG OF TRANS. BLOCK]*
24	18	11000	30	*[CANCEL]*
25	19	11001	31	*[END OF MEDIUM]*
26	1A	11010	32	*[SUBSTITUTE]*
27	1B	11011	33	*[ESCAPE]*
28	1C	11100	34	*[FILE SEPARATOR]*
29	1D	11101	35	*[GROUP SEPARATOR]*
30	1E	11110	36	*[RECORD SEPARATOR]*
31	1F	11111	37	*[UNIT SEPARATOR]*
32	20	100000	40	*[SPACE]*
33	21	100001	41	!
34	22	100010	42	"
35	23	100011	43	#
36	24	100100	44	$
37	25	100101	45	%
38	26	100110	46	&
39	27	100111	47	'
40	28	101000	50	(
41	29	101001	51	)
42	2A	101010	52	*
43	2B	101011	53	+
44	2C	101100	54	,
45	2D	101101	55	-
46	2E	101110	56	.
47	2F	101111	57	/
48	30	110000	60	**0**
49	31	110001	61	**1**
50	32	110010	62	**2**
51	33	110011	63	**3**
52	34	110100	64	**4**
53	35	110101	65	**5**
54	36	110110	66	**6**
55	37	110111	67	**7**
56	38	111000	70	**8**
57	39	111001	71	**9**
58	3A	111010	72	:
59	3B	111011	73	;
60	3C	111100	74	<
61	3D	111101	75	=
62	3E	111110	76	>
63	3F	111111	77	?
64	40	1000000	100	@
65	41	1000001	101	**A**
66	42	1000010	102	**B**
67	43	1000011	103	**C**
68	44	1000100	104	**D**
69	45	1000101	105	**E**
70	46	1000110	106	**F**
71	47	1000111	107	**G**
72	48	1001000	110	**H**
73	49	1001001	111	**I**
74	4A	1001010	112	**J**
75	4B	1001011	113	**K**
76	4C	1001100	114	**L**
77	4D	1001101	115	**M**
78	4E	1001110	116	**N**
79	4F	1001111	117	**O**
80	50	1010000	120	**P**
81	51	1010001	121	**Q**
82	52	1010010	122	**R**
83	53	1010011	123	**S**
84	54	1010100	124	**T**
85	55	1010101	125	**U**
86	56	1010110	126	**V**
87	57	1010111	127	**W**
88	58	1011000	130	**X**
89	59	1011001	131	**Y**
90	5A	1011010	132	**Z**
91	5B	1011011	133	[
92	5C	1011100	134	\
93	5D	1011101	135	]
94	5E	1011110	136	^
95	5F	1011111	137	_
96	60	1100000	140	`
97	61	1100001	141	**a**
98	62	1100010	142	**b**
99	63	1100011	143	**c**
100	64	1100100	144	**d**
101	65	1100101	145	**e**
102	66	1100110	146	**f**
103	67	1100111	147	**g**
104	68	1101000	150	**h**
105	69	1101001	151	**i**
106	6A	1101010	152	**j**
107	6B	1101011	153	**k**
108	6C	1101100	154	**l**
109	6D	1101101	155	**m**
110	6E	1101110	156	**n**
111	6F	1101111	157	**o**
112	70	1110000	160	**p**
113	71	1110001	161	**q**
114	72	1110010	162	**r**
115	73	1110011	163	**s**
116	74	1110100	164	**t**
117	75	1110101	165	**u**
118	76	1110110	166	**v**
119	77	1110111	167	**w**
120	78	1111000	170	**x**
121	79	1111001	171	**y**
122	7A	1111010	172	**z**
123	7B	1111011	173	{
124	7C	1111100	174	|
125	7D	1111101	175	}
126	7E	1111110	176	~
127	7F	1111111	177	*[DEL]*

File Size

- Given a text file we construct alphabet $A = \{a_0, a_1, \ldots, a_{n-1}\}$ of all symbols in the file.
- Let F denote the total number of symbols (characters) in the file.
- Let f_i denote the frequency of occurrence of symbol a_i in the file, $i = 0, 1, \ldots, n-1$.
- Then,

$$F = \sum_{i=0}^{n-1} f_i \, .$$

Probability of occurrence of symbol

Since symbol a_i occurs with frequency f_i the probability p_i that symbol a_i occurs in a given text position is

$$p_i = \frac{f_i}{\sum_{i=0}^{n-1} f_i}$$

Compression using Binary Coding

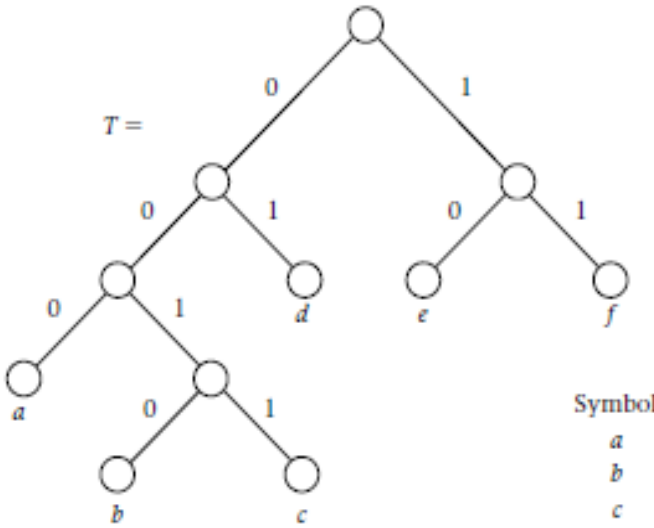
- If ASCII is used the size of the file is F bytes or $8F$ bits.
- We can do better than this by allowing variable length binary coding.
- For example, suppose $A = \{a, b, c\}$ and symbols a, b, c are encoded as 1, 00, 01, respectively.
- Then the binary encoding for the string *cabbc* is 011000001 having 9 bits as opposed to $5 \times 8 = 40$ bits using the ASCII encoding.

Ambiguous Codes

- Suppose *a*,*b*,*c* are encoded as 0, 1, 01, respectively.
- Then 011 can be decoded as either *abb* or *cb*.
- Note that 0 is a **prefix** of 01.
- It is easily verified that a binary code allows for **unambiguous** (unique) decoding if no binary string in the code is a prefix of any other binary string in the code.
- Binary codes having the property that no binary string in the code is a prefix of any other binary string in the code are called **prefix** *codes*.

Generating Prefix Codes using 2-trees

- 2-trees can be used to generate prefix codes.
- The leaf nodes of T correspond to the symbols in A.
- Label each edge corresponding to a left child 0 and right child 1.
- Assign to each symbol a_i the binary string obtained by reading the labels on the edges when following a path from the root to the leaf node corresponding to a_i .



Symbol	Binary encoding
a	000
b	0010
c	0011
d	01
e	10
f	11

Statement of Problem

- Let λ_i denote the length of a path in the 2-tree T from the root node to the leaf node containing the symbol a_i.
- Let $WLPL(T) = \sum_{i=0}^{n-1} \lambda_i \, f_i$.
- $WLPL(T)$ is called **the weighted leaf path length**.
- Each binary digit in the binary string encoding a_i corresponds to an edge of the path in T from the root node to the leaf node containing a_i.
- Hence, the length of the binary string encoding a_i equals $\lambda_i, i = 0, \ldots, n-1,$
- Thus, the expected length of a coded symbol is

$$\frac{WLPL(T)}{F}$$

Huffman Tree and Huffman Code

- An **optimal** binary code is one associated with 2-tree that minimizes the expected length of a symbol.
- Equivalently, the problem is to

$$\text{Minimize } WLPL(T)$$

over all 2-trees with symbols in leaf nodes having frequencies $f_i, i = 0, \ldots, n-1$.

- The optimizing tree is called a Huffman tree and the associate binary code is called Huffman code.

Greedy algorithm

Start with a forest consisting of leaf nodes.

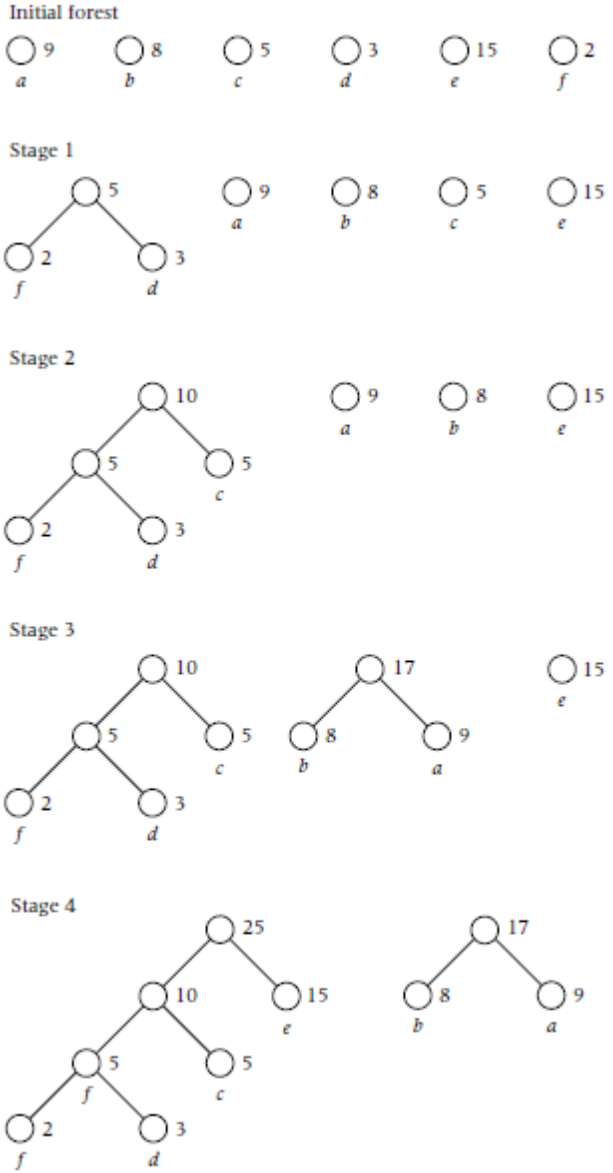
Repeat $n - 1$ times

Find two nodes c_1 and c_2 having smallest and second smallest frequencies

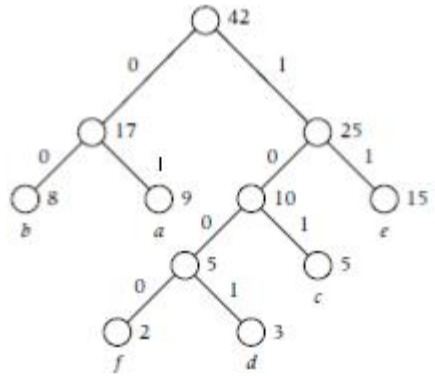
Create a new node p making c_1 its left child and c_2 its right child

Assign p the sum of the frequencies of c_1 and c_2

Illustration of greedy algorithm



Stage 5: Huffman tree



Symbol	Frequency	Encoding
a	9	01
b	8	00
c	5	101
d	3	1001
e	15	11
f	2	1000

$WLPL(T) = (2)(9) + (2)(8) + (3)(5) + (4)(3) + (2)(15) + (4)(2) = 99$

PSN. Illustrate greedy method for computing Huffman Tree and Huffman code following instance:

symbol	a	b	c	d	e	f	g
frequency	10	2	6	5	4	12	5

Pseudocode for Huffman Code

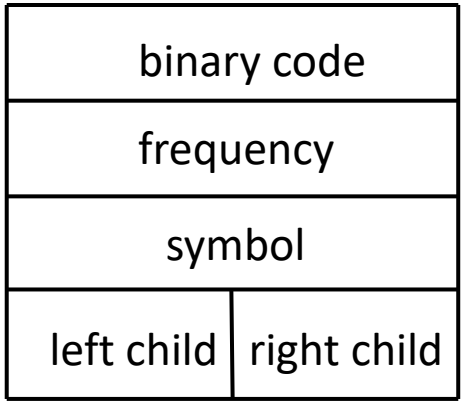
```
procedure HuffmanCode(Freq[0:n – 1],Code[0:n – 1])
Input: Freq[0:n – 1]    (an array of nonnegative frequencies: Freq[i] = f_i)
Output: Code[0:n – 1] (an array of binary strings for Huffman code: Code[i] is
                                the binary string encoding symbol a_i, i = 0,...,n – 1)
        for i ← 0 to n – 1 do  //initialize leaf nodes
                AllocateHuffmanNode(P)
                P → SymbolIndex ← i
                P → Frequency ← Freq[i]
                P → LeftChild ← null
                P → RightChild ← null
                Leaf[i] ← P
        endfor
        CreatePriorityQueue(Leaf[0:n – 1],Q)      //create priority queue of
              //pointers to leaf nodes with Frequency as the key
        for i ← 1 to n – 1 do
                RemovePriorityQueue(Q,L)          //L,R point to root nodes of
                                                  //smallest and
                RemovePriorityQueue(Q,R)          //second-smallest frequency,
                                                  //respectively
                AllocateHuffmanNode(Root)
                Root → LeftChild ← L
                Root → RightChild ← R
                Root → Frequency ← (L → Frequency) + (R → Frequency)
                InsertPriorityQueue(Q,Root)
         endfor
         Root → BinaryString ← ''     //BinaryString of root initialized to null
                                                                    //string
        GenerateCode(Root,Code[0:n – 1])
end HuffmanCode
```

GenerateCode is based on a preorder traversal of the 2-tree *T*.

```
procedure GenerateCode(Root,Code[0:n – 1]) recursive
Input:  Root (a pointer to root of 2-tree T)    //Root→BinaryCode is initialized to
                                                // the null string. The leaf node corresponding to a_i has its
                                                // SymbolIndex field initialized to i, i = 0,...,n – 1
Output: Code[0:n – 1] (array of binary strings, where Code[i] is the code for
                                                                                symbol a_i)
        if Root→LeftChild = null then          //a leaf is reached
                Code[Root→SymbolIndex] ← Root→BinaryCode
        else
                (Root→LeftChild)→BinaryCode ← Root→BinaryCode + '0'
                (Root→RightChild)→BinaryCode ← Root→BinaryCode + '1'
                GenerateCode(Root→LeftChild,Code)
                GenerateCode(Root→RightChild,Code)
        endif
end GenerateCode
```

Using Priority Queue to Implement

For each node of 2-tree we **dynamically allocate a node** (struct or class in C++) consisting of five fields, where the frequency represents the priority, a smaller frequency having a higher priority. The priority queue consists of pointers to (i.e., address of) these nodes.



To construct Huffman tree:

Repeat until priority queue is empty

1. allocate a new node *N*
2. dequeue and assign to left child of *N*
3. dequeue and assign to right child of *N*
4. assign frequency of *N* the sum of the frequencies of the two dequeues
5. if priority queue is not empty
   enqueue pointer to *N*

Sending compressed file

- When we send compressed text data with all symbols replaced with their Huffman binary code, we also need to send info so that the receiver can decode the data.
- It is not necessary to send the Huffman tree.
- Simply send the symbols and their associated frequencies and reconstruct the Huffman tree at the receiving end.
- The only caveat is that the Huffman tree needs to be constructed in a canonical way, so that the same tree is reconstructed at the other end.

Application to compression of image files

Huffman coding which we used to optimize compression of text, is applied in JPEG, which is a format for compressing image files.