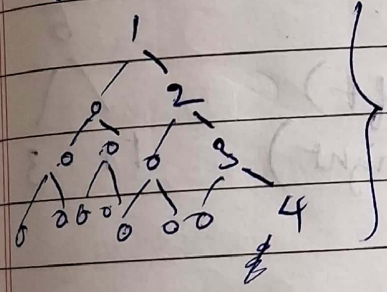
HW 12
Ruchika Shashidhara
NU 002245068.

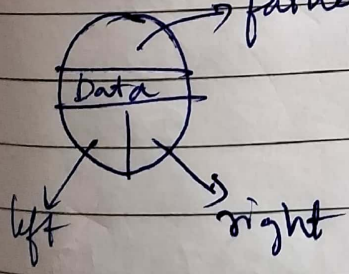
Why binary tree cannot be represented as an array?
All trees are not complete trees. if we store it as a python list or array - there will be empty spaces in the array - empty space

Ex: Worst case deep tree



would have required 15 (2^n spaces) spaces, but with node Struct requires 4 spaces (n spaces)

2) Need for left & right pointers?



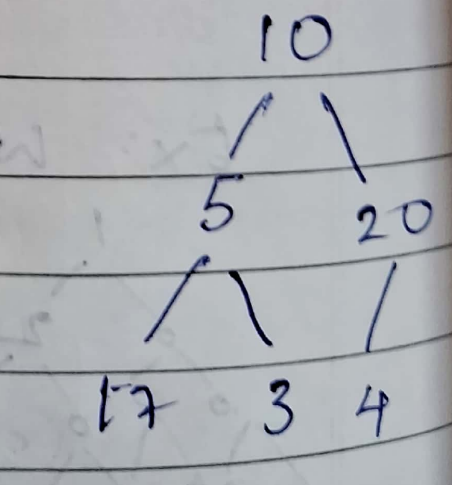
To efficiently store the tree we can use left ptr for left children & right ptr for right children we can also have father ptr (root has no father) 3) ~~root~~
here 3n spaces is required
can store non-complete binary trees efficiently.

3) Why parent pointer is not required?
We can back track to find the father node & we can store only 1 node additional pointer to the root of the tree. (2n+1 spaces are req

4) Tree traversal

5) Pre Order : VLR

```
order(node):
  if (node):
    visit(node)
    preorder(node.left)
    preorder(node.right)
```



preorder (5)
preorder (17)
visit (17)

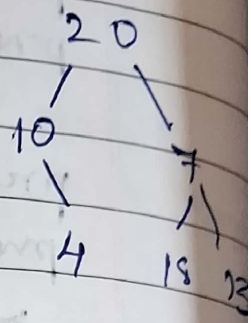
Stack trace:

		Current o/p
preorder (10)	① × ㉔	
visit (10)	② × ③	10
preorder (5)	④ × ⑮	
visit (5)	⑤ × ⑥	10, 5
preorder (17)	⑦ × ⑩	
visit (17)	⑧ × ⑨	10, 5, 17
preorder (3)	⑪ × ⑭	
visit (3)	⑫ × ⑬	10, 5, 17, 3
preorder (20)	⑭ × ⑰	
visit (20)	⑰ × ⑱	10, 5, 17, 3, 20
preorder (4)	⑲ × ㉒	
visit (4)	⑳ × ㉑	10, 5, 17, 3, 20, 4

∴ preorder: 10, 5, 17, 3, 20, 4

2) Inorder : L V R

```
inorder (node):
    if (node):
        inorder (node.left)
        visit (node)
        inorder (node.right)
```



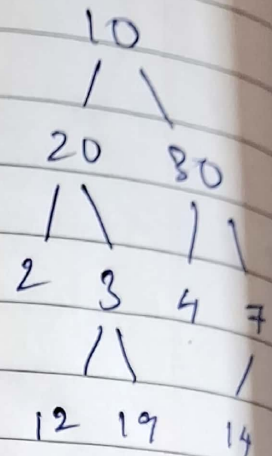
Stack Trace:

			current o/p
inorder (20)	(1)	(23)	
inorder (10)	(2)	(8)	
visit (10)	(3)	(4)	10
inorder (4)	(5)	(8)	
visit (4)	(6)	(7)	10, 4
visit (20)	(9)	(10)	10, 4, 20
inorder (7)	(11)	(22)	
inorder (18)	(12)	(15)	
visit (18)	(13)	(14)	10, 4, 20, 18
visit (7)	(16)	(17)	10, 4, 20, 18, 7
inorder (13)	(18)	(21)	
visit (13)	(19)	(20)	10, 4, 20, 18, 7, 13

∴ Inorder : 10, 4, 20, 18, 7, 13

Post order : L R V

```
postorder(node):
    if(node):
        postorder(node.left)
        postorder(node.right)
        visit(node)
```



Stack Trace:

		Current o/p
postorder(10)	① × ㊱	
postorder(20)	② × ⑲	
postorder(2)	③ × ⑥	
visit(2)	④ × ⑤	
postorder(3)	⑦ × ⑰	
postorder(12)	⑧ × ⑪	
~~postorder(~~		
visit(12)	⑨ × ⑩	2, 12
postorder(19)	⑫ × ⑮	
visit(19)	⑬ × ⑭	2, 12, 19
visit(3)	⑮ × ⑯	2, 12, 19, 3
visit(20)	⑰ × ⑱	2, 12, 19, 3, 20
postorder(30)	⑳ × ㉞	
postorder(4)	㉑ × ㉔	
visit(4)	㉒ × ㉓	2, 12, 19, 3, 20, 4
postorder(7)	㉕ × ㉜	
postorder(14)	㉖ × ㉙	
visit(14)	㉗ × ㉘	2, 12, 19, 3, 20, 4, 14

visit(7) (30) × (31) 2, 12, 19, 3, 20, 4, 14, 7

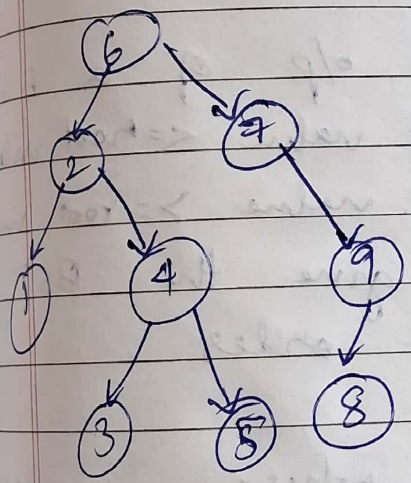
visit(30) (32) × (33) 2, 12, 19, 3, 20, 4, 14, 7, 30

visit(10) (30) × (38) 2, 12, 19, 3, 20, 4, 14, 7, 30, 10

∴ postorder: 2, 12, 19, 3, 20, 4, 14, 7, 30, 10

Level-Order Traversal

Printing in the same line:



Time complexity: $\Theta(N)$

Space Complexity = $\Theta(N)$

q → 6

q → 2, 7

q → 1, 4, 9

q → 3, 5, 8

q →

Output:

6

6, 2

6, 2, 7

6, 2, 7, 1

6, 2, 7, 1, 4

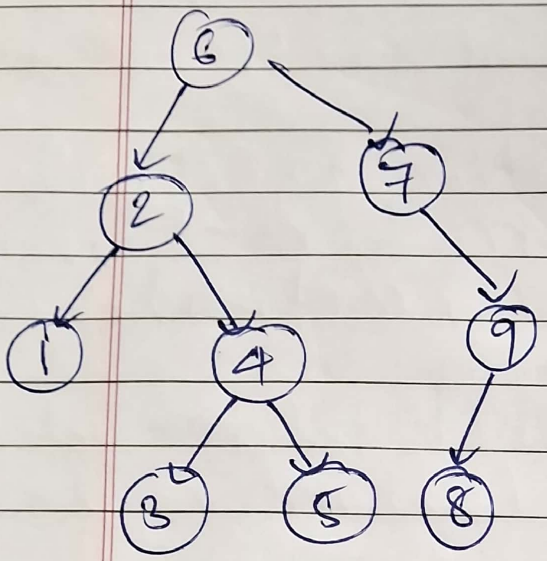
6, 2, 7, 1, 4, 9

6, 2, 7, 1, 4, 9, 3

6, 2, 7, 1, 4, 9, 3, 5

6, 2, 7, 1, 4, 9, 3, 5, 8

Level Order Traversal - BFS



q → 6 in a single line

print : level order

6 2 7 1 4 9
3 5 8

q → ~~6~~ ~~2~~ ~~7~~ ~~1~~ ~~4~~ ~~9~~ ~~3~~ ~~5~~ ~~8~~

N = No. of Nodes

Time complexity = $\Theta(N)$

Space complexity = $\Theta(N)$

q → 6 N → it is a differentiator node, can be None or Null in Python

bfs →

q → N 2 7

bfs → 6

Time complexity $\approx \Theta(N)$

Space complexity $\approx \Theta(N)$

q → 2 7 N

bfs → 6 \n

q → 7 N 1 4

bfs → 6 \n 2

q → N 1 4 9

bfs → 6 \n 2 7

q → 1 4 9 N
bfs → 6 \n 2 7 \n

q → 4 9 N
bfs → 6 \n 2 7 \n 1

q → 9 N 3 5
bfs → 6 \n 2 7 \n 1 4

q → N 3 5 8
bfs → 6 \n 2 7 \n 1 4 9

q → 3 5 8 N
bfs → 6 \n 2 7 \n 1 4 9 \n

q → 5 8 N
bfs → 6 \n 2 7 \n 1 4 9 \n 3

q → 8 N
bfs → 6 \n 2 7 \n 1 4 9 \n 3 5

q → N
bfs → 6 \n 2 7 \n 1 4 9 \n 3 5 8

q →
bfs → 6 \n 2 7 \n 1 4 9 \n 3 5 8 \n

∴ Print:
6
2 7
1 4 9
3 5 8

} In multiple lines acc. to level order

6) Give an example where postorder is required

Use case: Deleting all nodes of a tree

∴ First child nodes need to be deleted before deleting the parent node.

7) Give an example where inorder traversal is required

Use case: Sorted o/p of a BST

∴ In a BST left.value <= root.value & right.value >= root.value

so inorder will give the BST in sorted Asc order.

8) Give an example where preorder traversal is required.

Use case: In AST - Abstract Syntax Tree → Expression Eval

∴ when we want to evaluate an expression & leaves are numbers or operands & internal nodes are operators.

Huffman Encoding

a = 0000
b = 0101
c = 0100
d = 1110
e = 1101
f = 1100

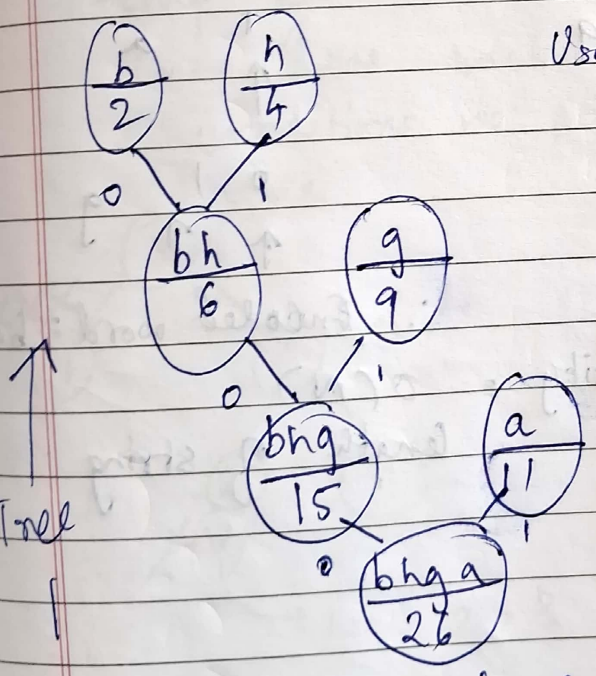
char	freq
a	11
b	2
g	9
h	4

ASCII takes 8 chars

Given a string
Ex: Northeastern
How many times does each char occur?

n = 2 e = 2
o = 1 a = 1
r = 2 s = 1
t = 2
h = 1

Use Cases: Zip Encryption



→ this can be implemented using heap

All leaves are chars

a = 1
b = 000
h = 001
g = 01

a g h
1 01 001

a a g
1 1 01

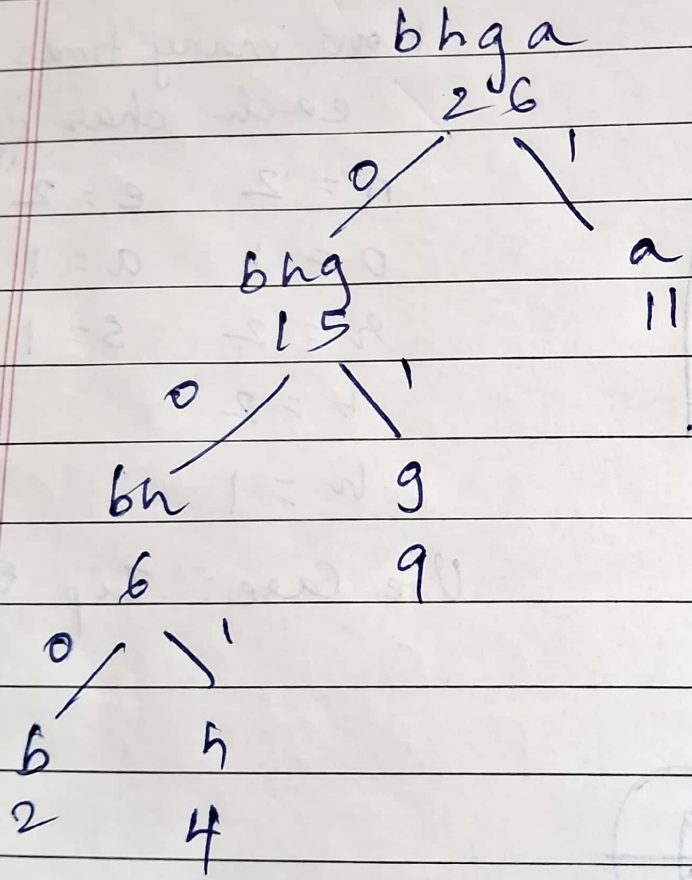
g g h a a g
01 01 001 1 1 01

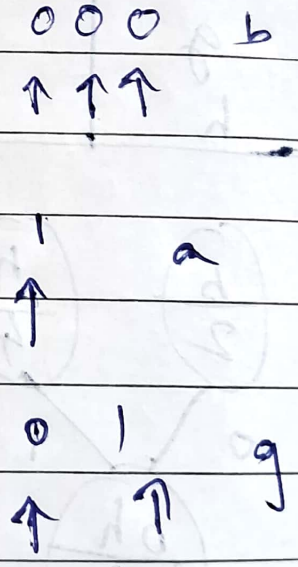
15)

000101

b = 000
h = 001
g = 01
a = 1

- Start from the root & keep traversing until string matches path.





∴ Encoded word = bag

Time complexity = O(N)
N = length of string.

char	freq
a	1
e	2
n	2
o	1
r	2
t	2
h	1
s	1

Northeastern or

```
count char freq (s):
    f = dict()
    for c in s:
        if c in f:
            f[c] = f[c] + 1
        else:
            f[c] = 0
    return f
```

Time: $\Theta(N)$
Space: $\Theta(N)$

$a_1 \; o_1 \; h_1 \; s_1 \; e_2 \; n_2 \; r_2 \; t_2$

$h_1 \; s_1 \; ao_2 \; e_2 \; n_2 \; r_2 \; t_2$

ao_2: $a_1 \; o_1$

$ao_2 \; hs_2 \; e_2 \; n_2 \; r_2 \; t_2$

hs_2: $h_1 \; s_1$

$e_2 \; n_2 \; r_2 \; t_2 \; aohs_4$

$aohs_4$: $ao_2 \; hs_2$

ao_2: $a_1 \; o_1$

hs_2: $h_1 \; s_1$

cn_4: $e_2 \; n_2$

$aohs_4$: $ao_2 \; hs_2$

ao_2: $a_1 \; o_1$

hs_2: $h_1 \; s_1$

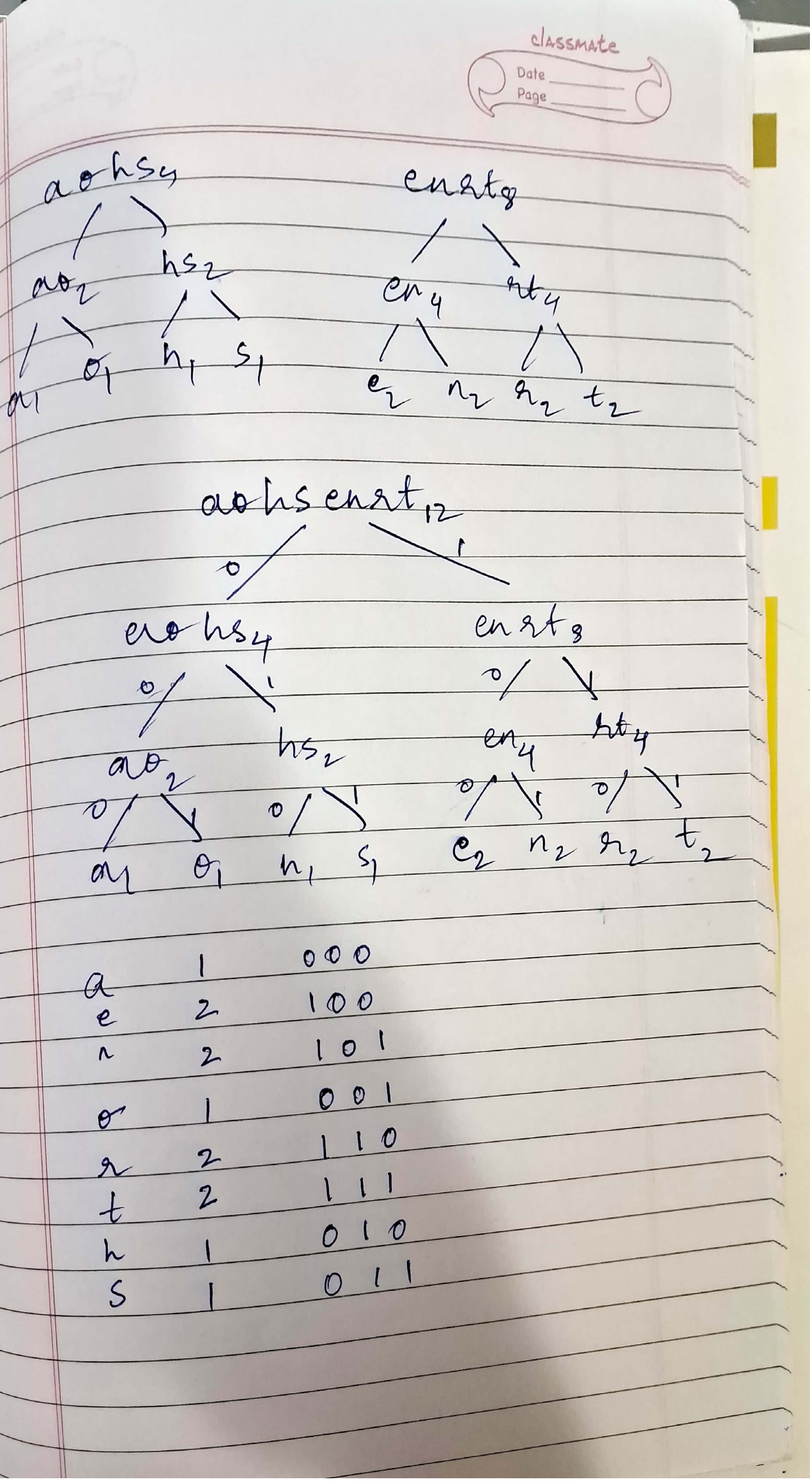
cn_4: $e_2 \; n_2$

rt_4: $r_2 \; t_2$

$aohs_4$: $ao_2 \; hs_2$

ao_2: $a_1 \; o_1$

hs_2: $h_1 \; s_1$



a	1	0 0 0
e	2	1 0 0
n	2	1 0 1
o	1	0 0 1
r	2	1 1 0
t	2	1 1 1
h	1	0 1 0
s	1	0 1 1

build tree:

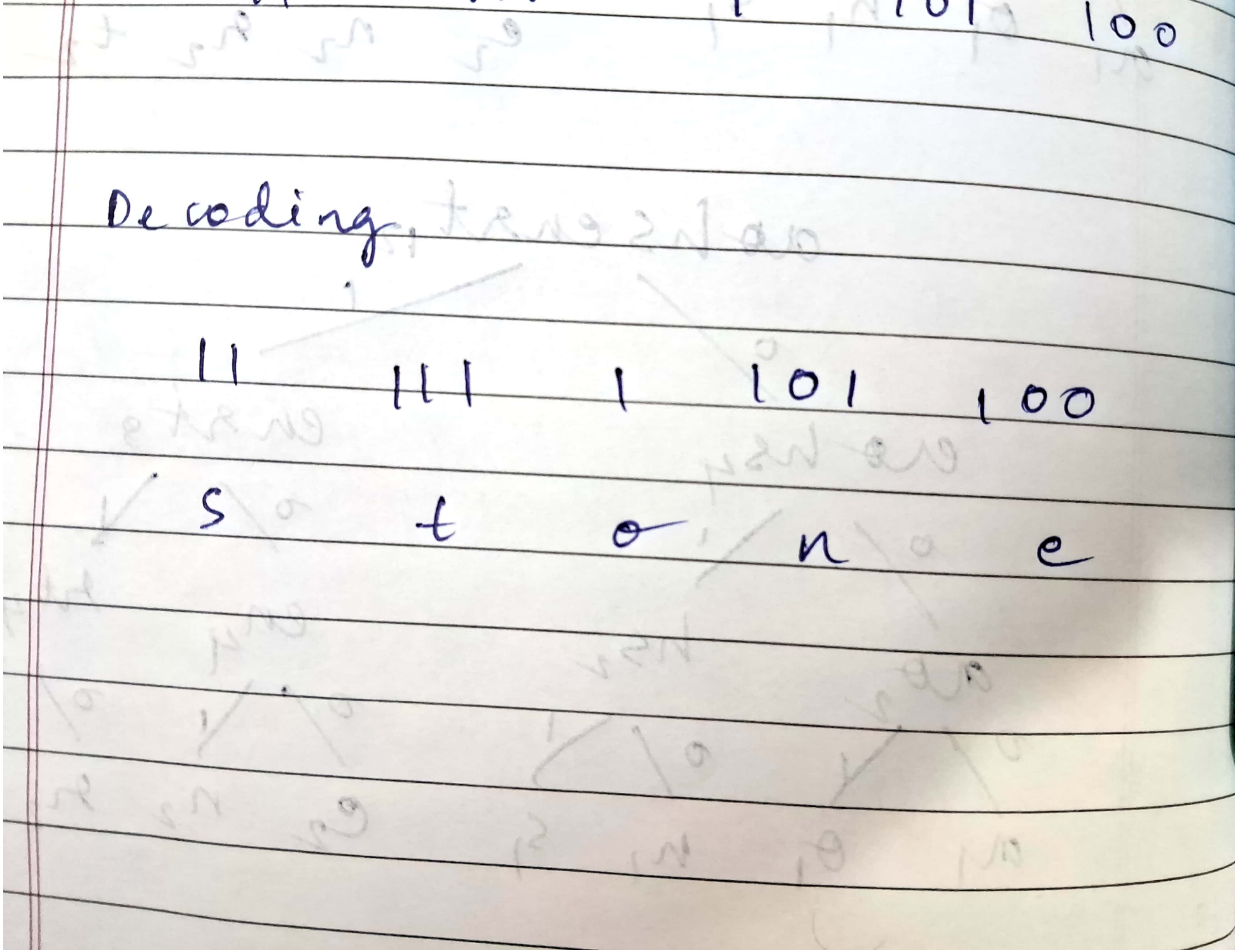
```
while len(heap) > 1:
    f1, left = heapq.pop(heap)
    f2, right = heapq.pop(heap)
    merged = [f1 + f2, [left, right]]
    heapq.heappush(heap, merged)
```

Generate codes:

```
def gen(node, prefix, codes):
    if isinstance(node, str):
        codes[node] = prefix
    else:
        gen(node[0], prefix + '0', codes)
        gen(node[1], prefix + '1', codes)
    return codes
```

Decode str:

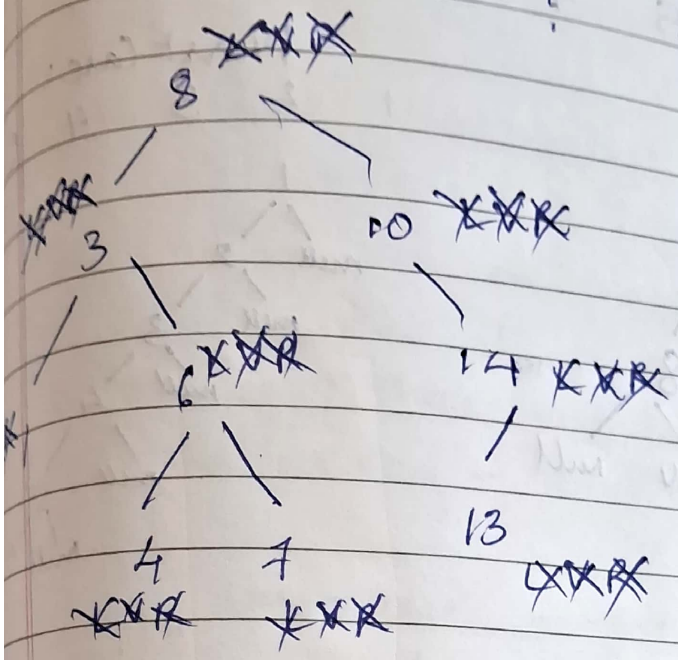
```
def decode(str, codes):
    revcodes = {code: c for c, code in
                codes.items()}

    decoded = ''
    curr = ''
    for bit in str:
        curr += bit
        if curr in revcodes:
            decoded += revcodes[curr]
            curr = ''
    return decoded
```


101 100
Decoding.
11 111 1 101 100
s t o n e

BST - Binary search Tree

Is tree a BST ?

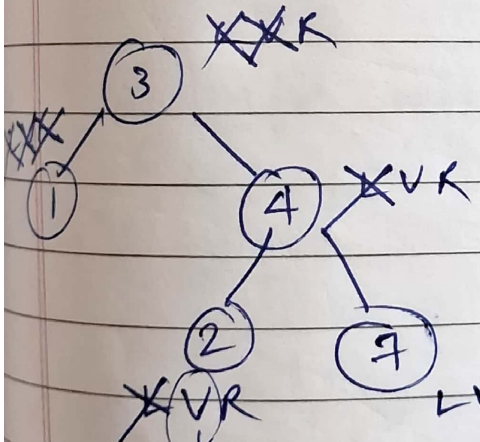
N -> no. of nodes




```
isBst(node, temp)
if node:
    isBst(node.left)
    isValid(node, temp)
    isBst(node.right)
isValid(node, temp)
    return (node.val > temp)

continue if:
√ node.val > temp
```

temp ~~0~~ 1 3 4 6 7 8 10 13 14

we came back to root, all conditions we passed ∴ It is a BST takes := N steps



temp 0
1
3

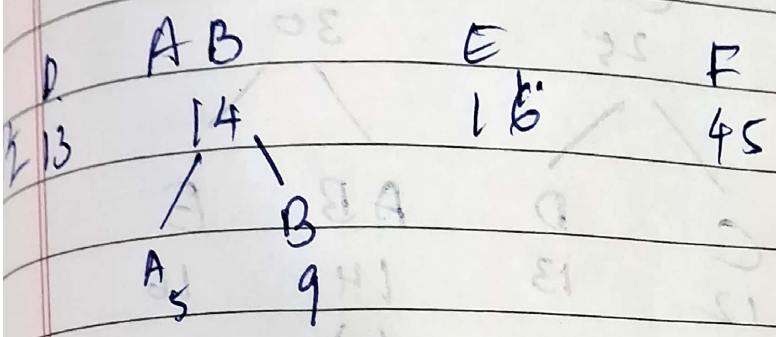
here 2 > 3 is false, condition fails, ∴ It is not a BST

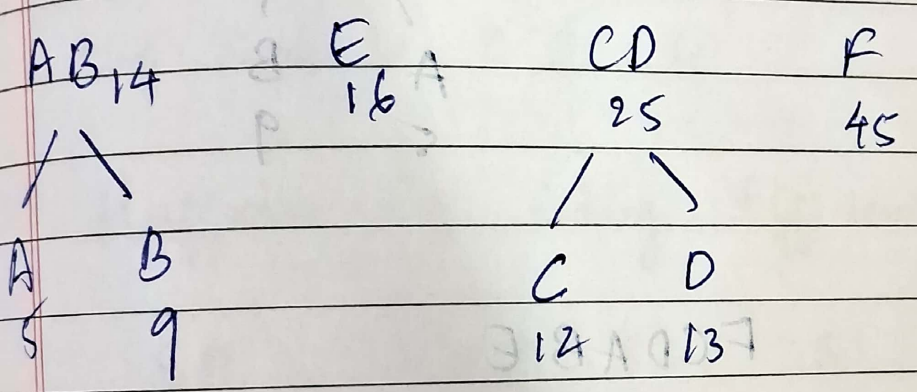
takes < N steps.

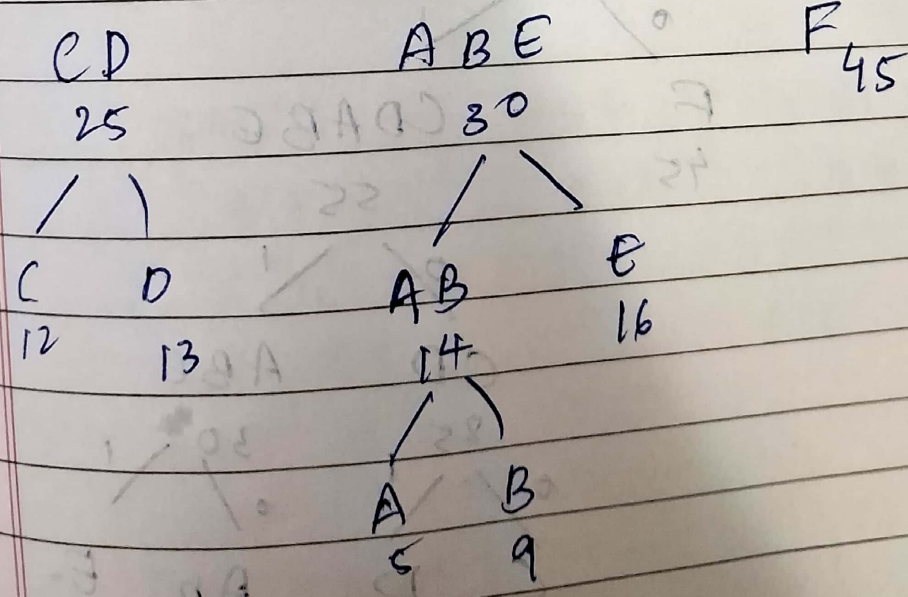
∴ Time complexity := O(N)

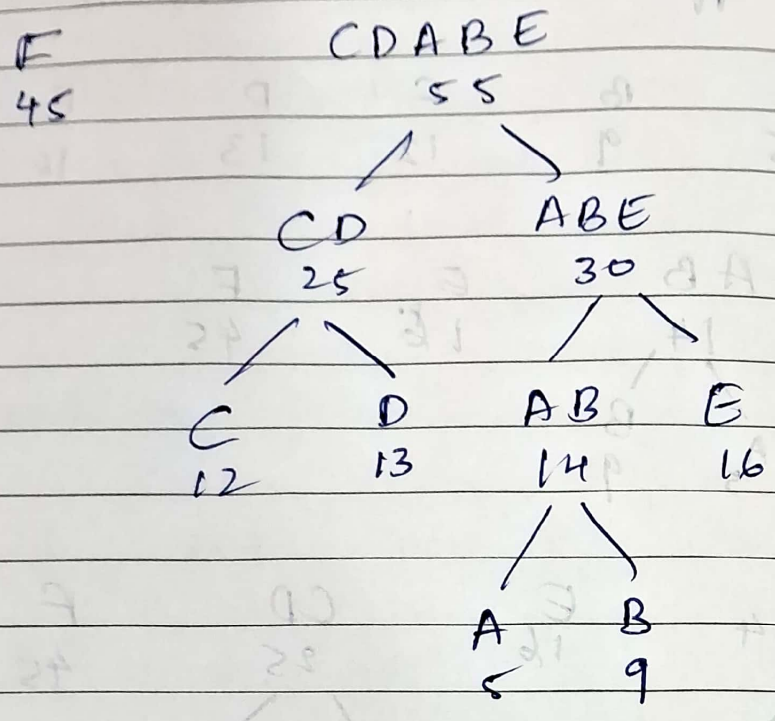
1) Huffman Tree Construction

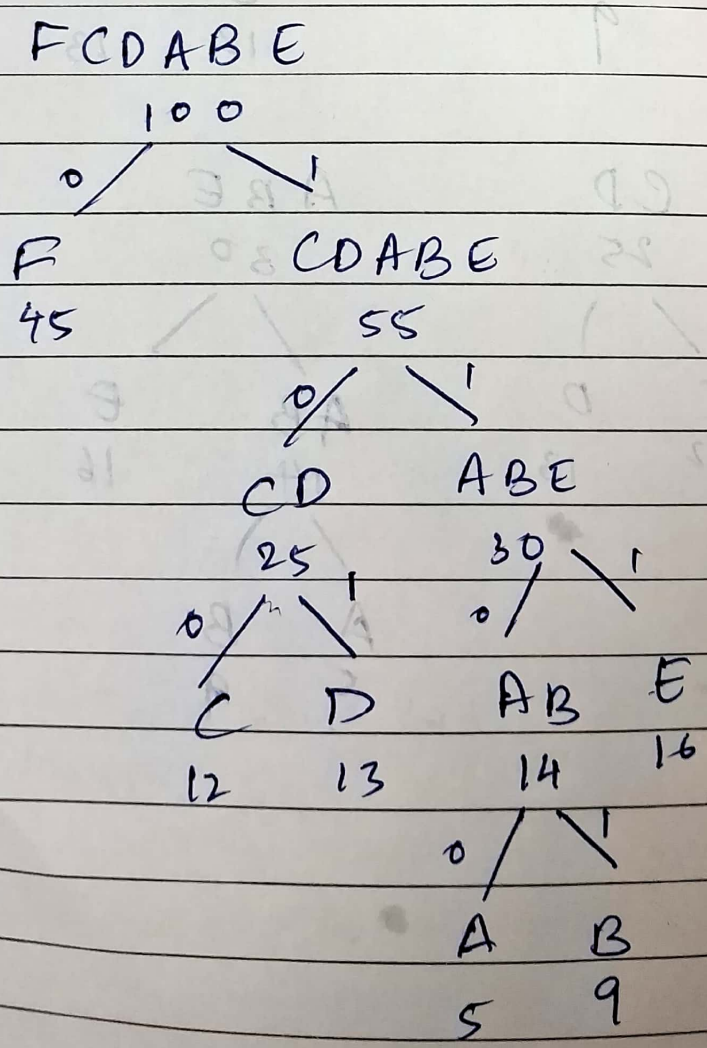
A	B	C	D	E	F
5	9	12	13	16	45











Symbol	Freq	Code	Len	Total bits
A	5	1100	4	20
B	9	1101	4	36
C	12	100	3	36
D	13	101	3	39
E	16	111	3	48
F	45	0	1	45
	100			224

∴ Total no. of bits required = 224

∴ Avg. code len = 2.24

2) Huffman Encoding Efficiency

S4 S3 S2 S1

0.1 0.2 0.3 0.4

S4 S3 S2 S1

0.3 0.3 0.4

S4 S3

0.1 0.2

S1 S4 S3 S2

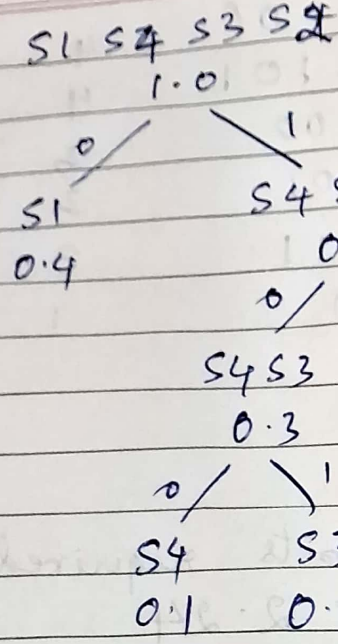
0.4 0.6

S4 S3 S2

0.3 0.3

S4 S3

0.2



Symbol	probability	encoding	Len	total bits (probability × Len)
S1	0.4	0	1	0.4
S2	0.3	11	2	0.6
S3	0.2	101	3	0.6
S4	0.1	100	3	0.3
				+ 1.9

code length = $\sum$ probability × len.

$= 0.4 \times 1 + 0.3 \times 2 + 0.2 \times 3 + 0.1 \times 3$

avg code length = 1.9

entropy $= -\sum$ prob $\times \log_2$ (prob)

$= -(0.4 \times \log_2 0.4 + 0.3 \times \log_2 0.3 + 0.2 \times \log_2 0.2 + 0.1 \times \log_2 0.1)$

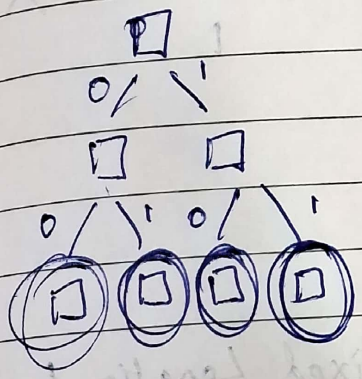
∴ entropy = −1.8464 bits

efficiency = entropy / code len = −1.8464 / 1.9

∴ efficiency = 97.18 %

3) Unique Decoding Property

A)



Given Found

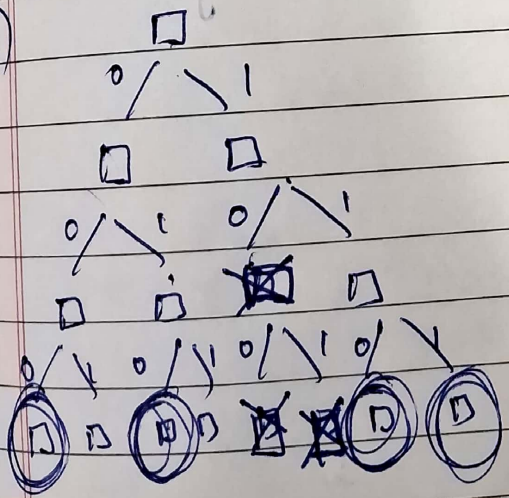
00 00 ✓

01 01 ✓

10 10 ✓

11 11 ✓ valid

B)



Given Found

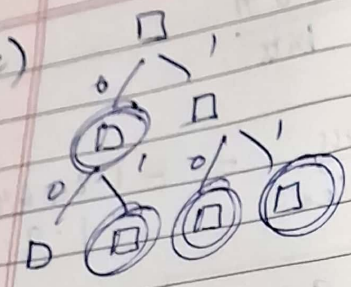
0 000 ✓

10 010 ✓

110 110 ✓

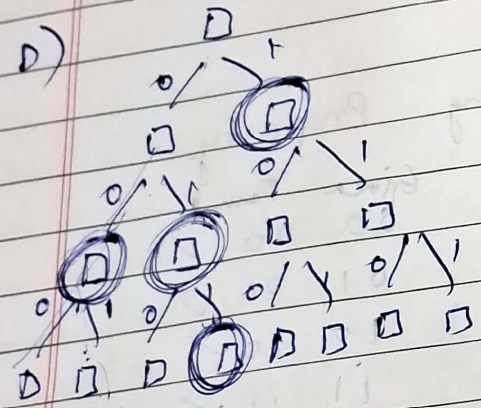
111 111 ✓ valid

c)



Given	Found
0	
01	X
10	invalid
11	

d)

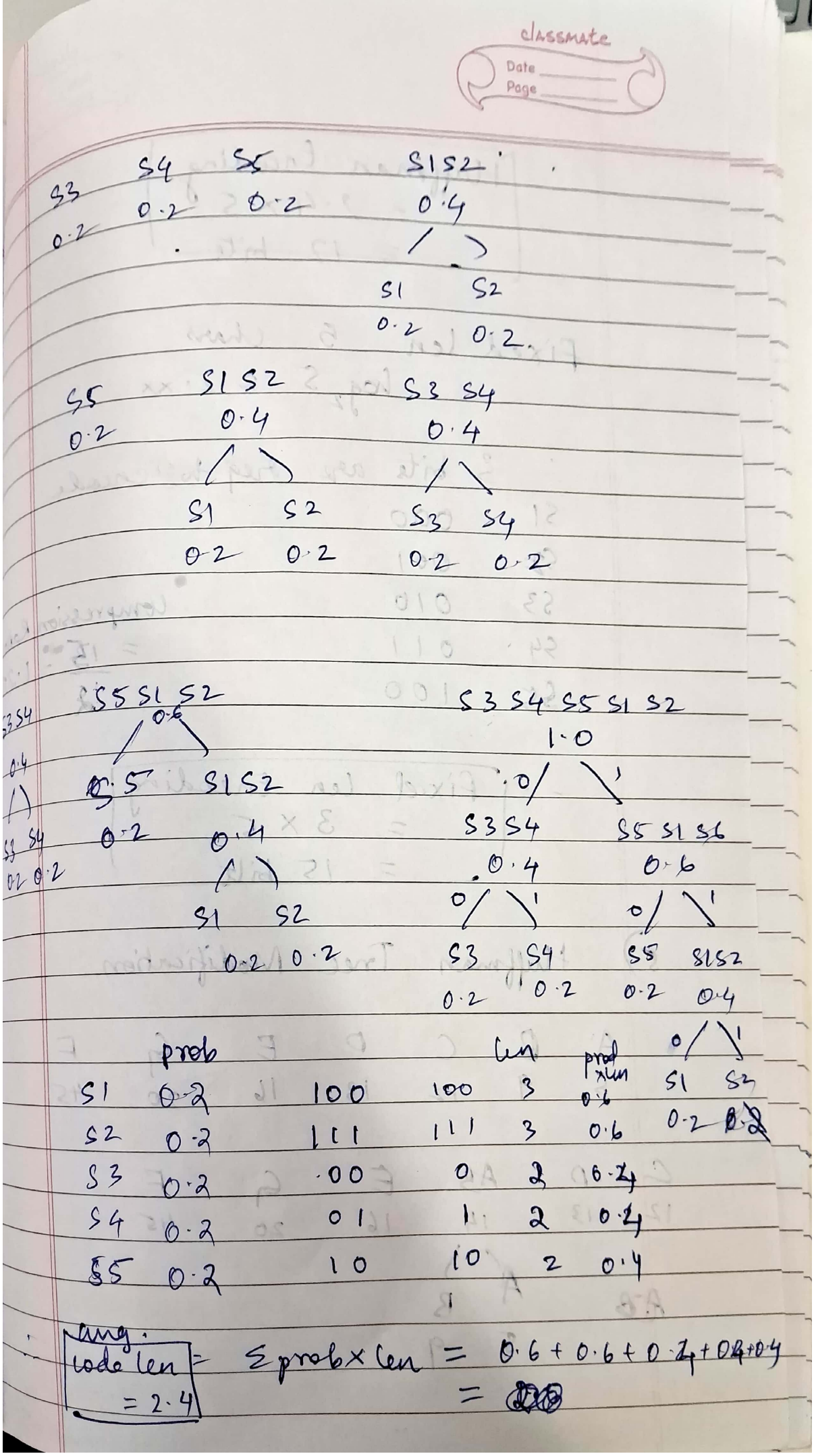


Given	Found
00	
01	
011	X
1	invalid

4) Huffman v/s Fixed Length Encoding

S1	0.2
S2	0.2
S3	0.2
S4	0.2
S5	0.2

S1	S2	S3	S4	S5
0.2	0.2	0.2	0.2	0.2



	prob		len	prod xlen	
S1	0.2	100	3	0.6	
S2	0.2	111	3	0.6	
S3	0.2	00	2	0.4	
S4	0.2	01	2	0.4	
S5	0.2	10	2	0.4	

avg. code len = Σ prob x len = 0.6 + 0.6 + 0.4 + 0.4 + 0.4

= 2.4

∴ Huffman Encoding
= 2.4 × 5
= 12 bits.

Fixed len 5 chars
$\log_2 5 \approx 2.xx$

∴ 3 bits are req to encode

S1	000
S2	001
S3	010
S4	011
S5	100

Compression Ratio
$= \frac{15}{12} = 1.25$

∴ Fixed len Encoding
= 3 × 5
= 15 bits

5) Huffman Tree Modification

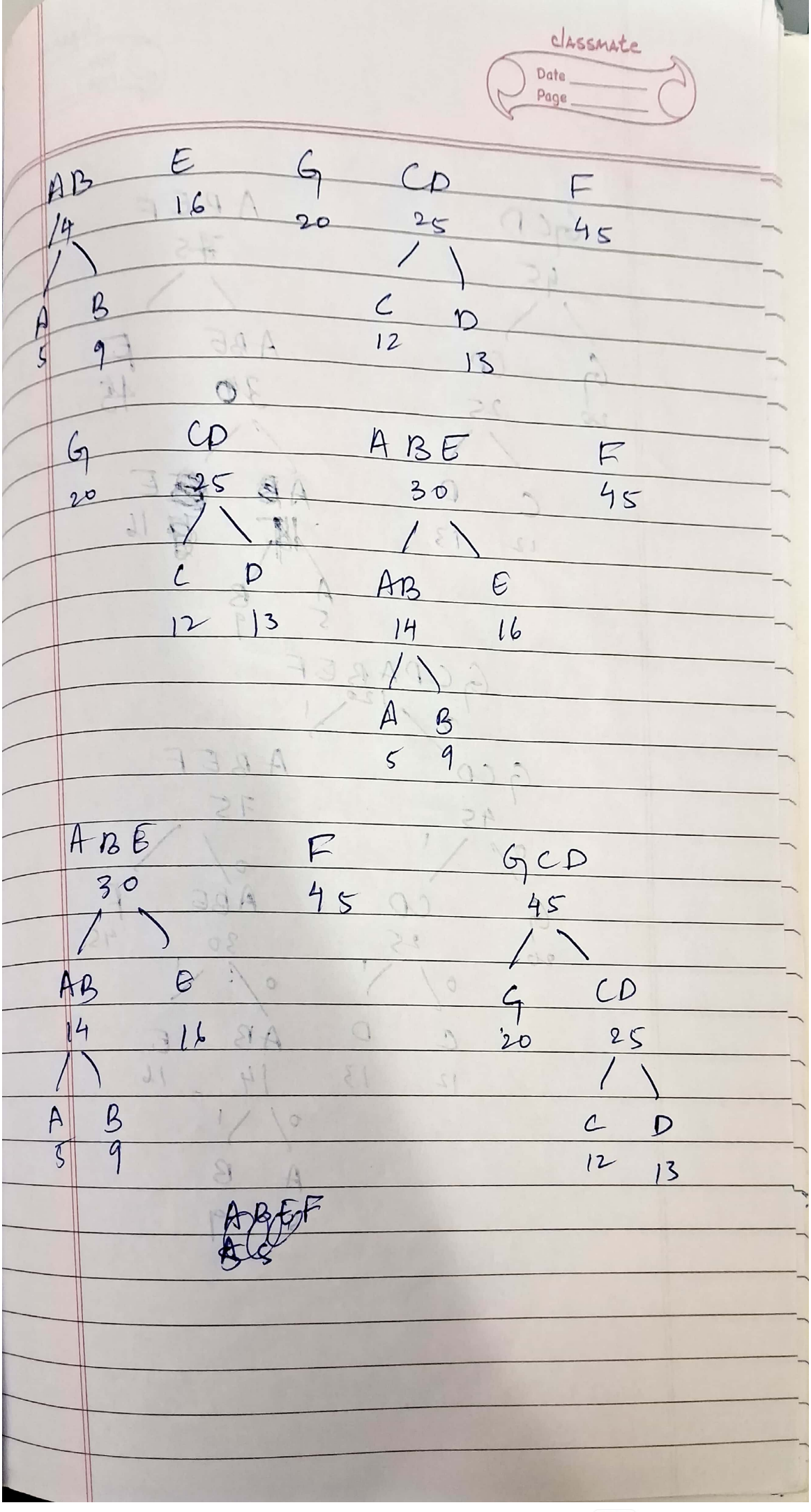
A	B	C	D	E	G	F
5	9	12	13	16	20	45

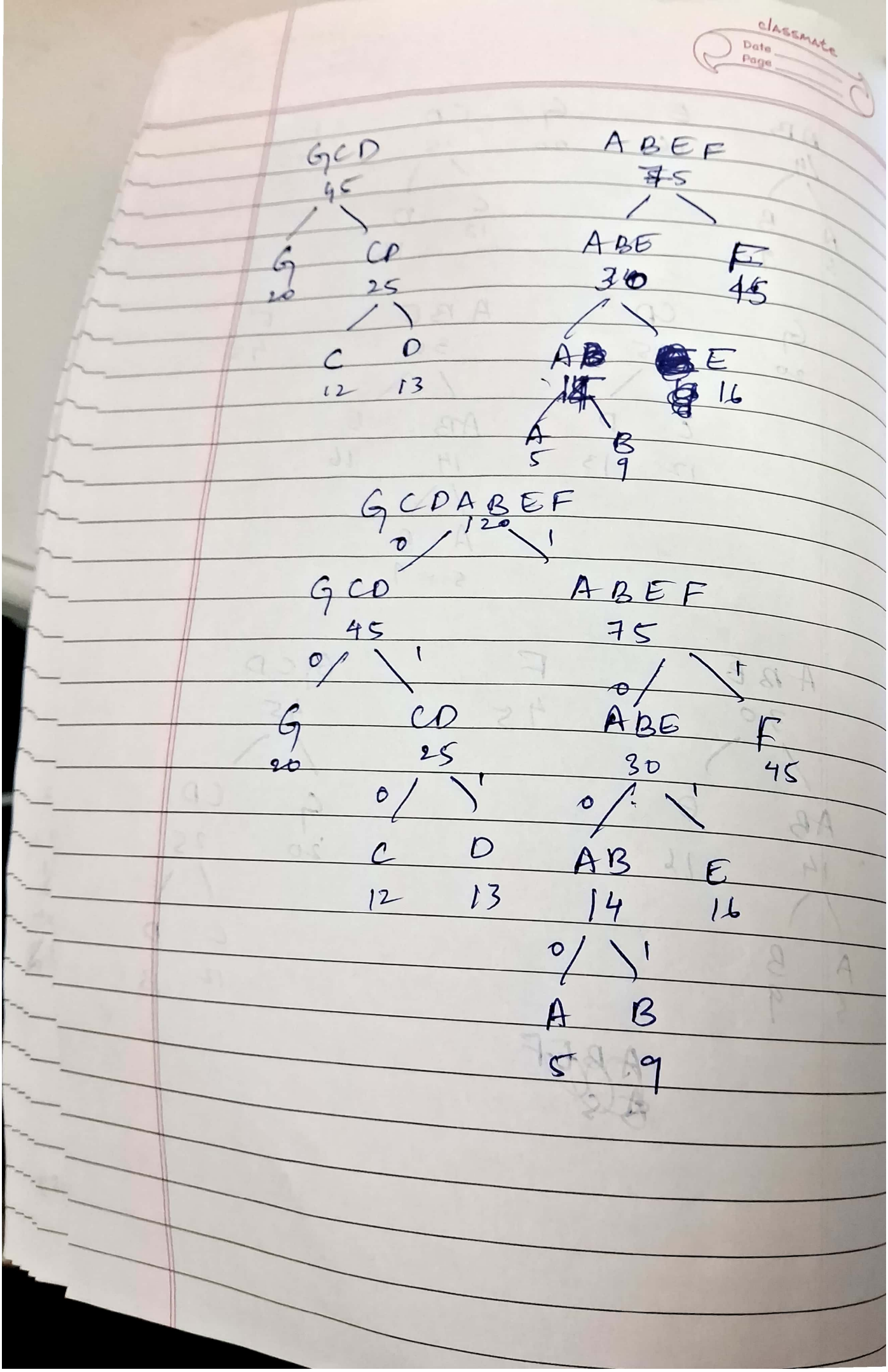
C	D	AB	E	G	F
12	13	14	16	20	45

AB

A B

5 9


AB
14
A
5
B
9
E
16
G
20
CD
25
C
12
D
13
F
45
G
20
CD
25
C
12
D
13
ABE
30
AB
14
A
5
B
9
E
16
F
45
ABE
30
AB
14
A
5
B
9
E
16
F
45
GCD
45
G
20
CD
25
C
12
D
13
ABEF


GCD
45
G
20
CD
25
C
12
D
13
ABEF
75
ABE
30
F
45
AB
14
E
16
A
5
B
9
GCDABEF
120
0
1
GCD
45
0
1
G
20
CD
25
0
1
C
12
D
13
ABEF
75
0
1
ABE
30
F
45
0
1
AB
14
E
16
0
1
A
5
B
9

prob	Symbol	Freq	Code	Len	Total Bits
.0416	A	5	1000	4	20
.0750	B	9	1001	4	36
.1000	C	12	010	3	36
.1083	D	13	011	3	39
.1333	E	16	101	3	48
.1666	G	20	00	2	40
.3750	F	45	11	2	90
		+ 120			+ 309

avg code len = $\sum$ prob × len = $\frac{\sum \text{freq} \times \text{len}}{\sum \text{freq}}$

= 0.0416 × 4 = 2.575 avg code len

+ 0.0750 × 4

+ 0.1000 × 3

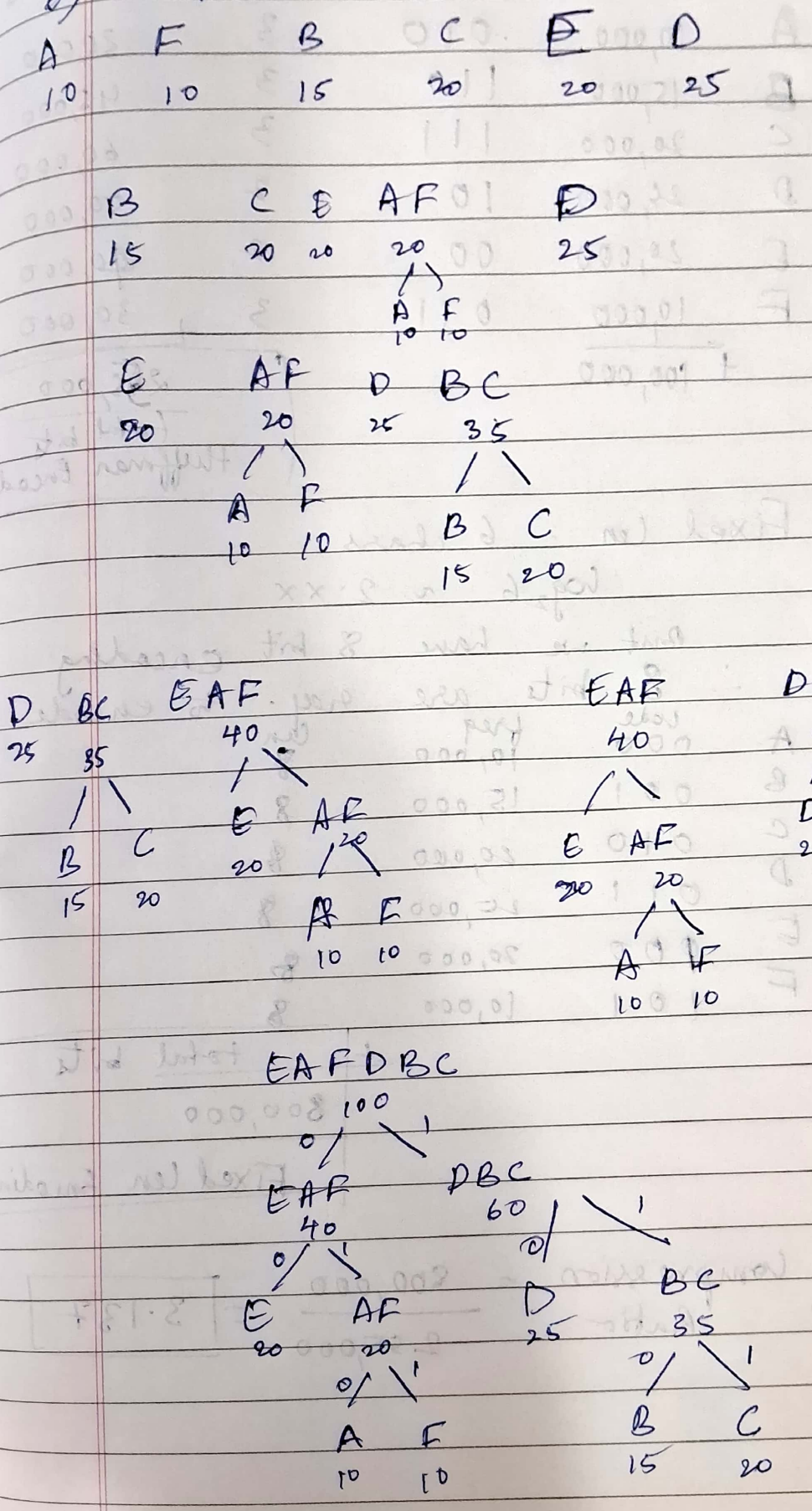
+ 0.1083 × 3

+ 0.1333 × 3

+ 0.1666 × 2

+ 0.3750 × 2

6) - Compression Ratio Calculation



Symbol	freq	code	len	total bits
A	10,000	010	3	30,000
B	15,000	110	3	45,000
C	20,000	111	3	60,000
D	25,000	10	2	50,000
E	20,000	00	2	40,000
F	10,000	011	3	30,000
	+ 100,000			+ 255,000

Total bits Huffman Encoding

Fixed Len : 6 chars

$\log_2 6 \approx 2.xx$

But we have 8 bit encoding

∴ 8 bits are req. to encode

	code	freq	len
A	000	10,000	8
B	001	15,000	8
C	010	20,000	8
D	011	25,000	8
E	100	20,000	8
F	101	10,000	8

+ total bits
800,000

Fixed Len Encoding

∴ Compression Ratio $= \frac{800,000}{255,000} = \boxed{3.137}$