Huffman Codes

Mashiwat Tabassum Waishy

Lecturer

Department *of*

CSE

STAMFORD UNIVERSITY BANGLADESH

Huffman Codes

- Huffman codes are an effective technique of ‘lossless data compression’ which means no information is lost.
- The algorithm builds a table of the frequencies of each character in a file.
  - The table is then used to determine an optimal way of representing each character as a binary string.

Huffman Codes

- Consider a file of 100,000 characters from a- f, with these frequencies:
  - a = 45,000
  - b = 13,000
  - c = 12,000
  - d = 16,000
  - e = 9,000
  - f = 5,000

Huffman Codes

- Typically each character in a file is stored as a single byte (8 bits)
  - If we know we only have six characters, we can use a 3 bit code for the characters instead:
    - a = 000, b = 001, c = 010, d = 011, e = 100, f = 101
  - This is called a fixed-length code
    - With this scheme, we can encode the whole file with 300,000 bits (45000*3+13000*3+12000*3+16000*3+9000*3+5000*3)
    - We can do better
      - Better compression
      - More flexibility

Huffman Codes

- Variable length codes can perform significantly better
  - Frequent characters are given short code words, while infrequent characters get longer code words
    - Consider this scheme:
      - a = 0; b = 101; c = 100; d = 111; e = 1101; f = 1100
    - How many bits are now required to encode our file?
      - 45,000*1 + 13,000*3 + 12,000*3 + 16,000*3 + 9,000*4 + 5,000*4
        = 224,000 bits
    - This is in fact an optimal character code for this file

Huffman Codes

- Prefix codes
  - Huffman codes are constructed in such a way that they can be unambiguously translated back to the original data, yet still be an optimal character code
  - Huffman codes are really considered *"prefix codes"*
    - No code word is a prefix of any other code word

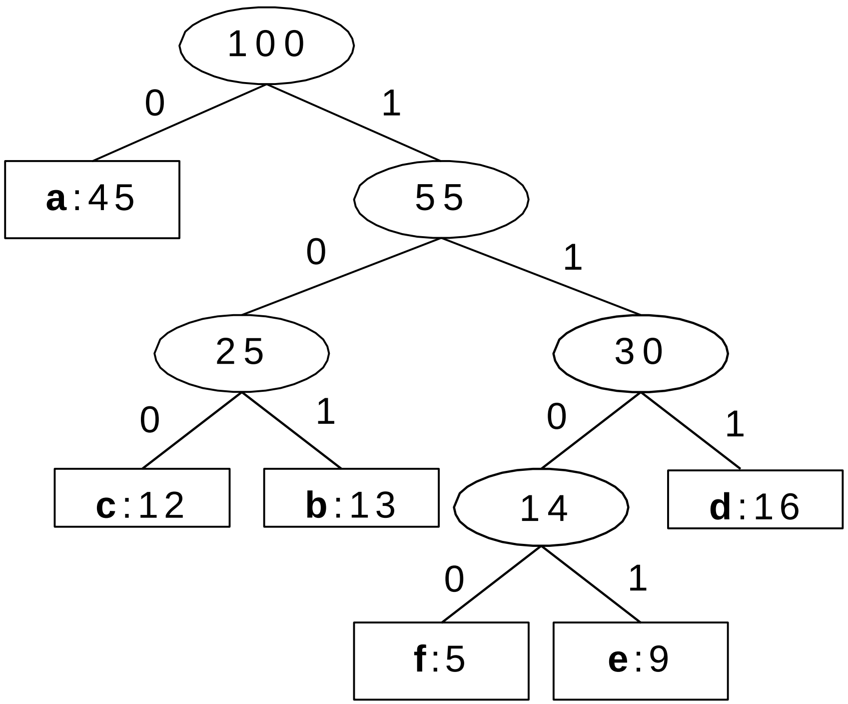
    Prefix code (9,55,50)

    Not a prefix code (9,5,55,59) because 5 is present in 55 and 59
    - This guarantees unambiguous decoding
      - Once a code is recognized, we can replace with the decoded data, without worrying about whether we may also match some other code.

Huffman Codes

- Both the encoder and decoder make use of a binary tree to recognize codes.
  - The leaves of the tree represent the unencoded characters
  - Each left branch indicates a “0” placed in the encoded bit string
  - Each right branch indicates a “1” placed in the bit string

Huffman Codes



A Huffman Code Tree

- To encode:
  - Search the tree for the character to encode
  - As you progress, add “0” or “1” to right of code
  - Code is complete when you find character
- To decode a code:
  - Proceed through bit string left to right
  - For each bit, proceed left or right as indicated
  - When you reach a leaf, that is the decoded character

Huffman Codes

- Using this representation, an optimal code will always be represented by a *full* binary tree
  - Every non-leaf node has two children
    - If this were not true, then there would be "waste" bits, as in the fixed-length code, leading to a non-optimal compression
  - For a set of *c* characters, this requires *c* leaves, and *c*-1 internal nodes

Huffman Codes

- Given a Huffman tree, how do we compute the number of bits required to encode a file?
  - For every character *c*:
    - Let $f(c)$ denote the character's frequency
    - Let $d_T(c)$ denote the character's depth in the tree
      - This is also the length of the character's code word
    - The total bits required is then:

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

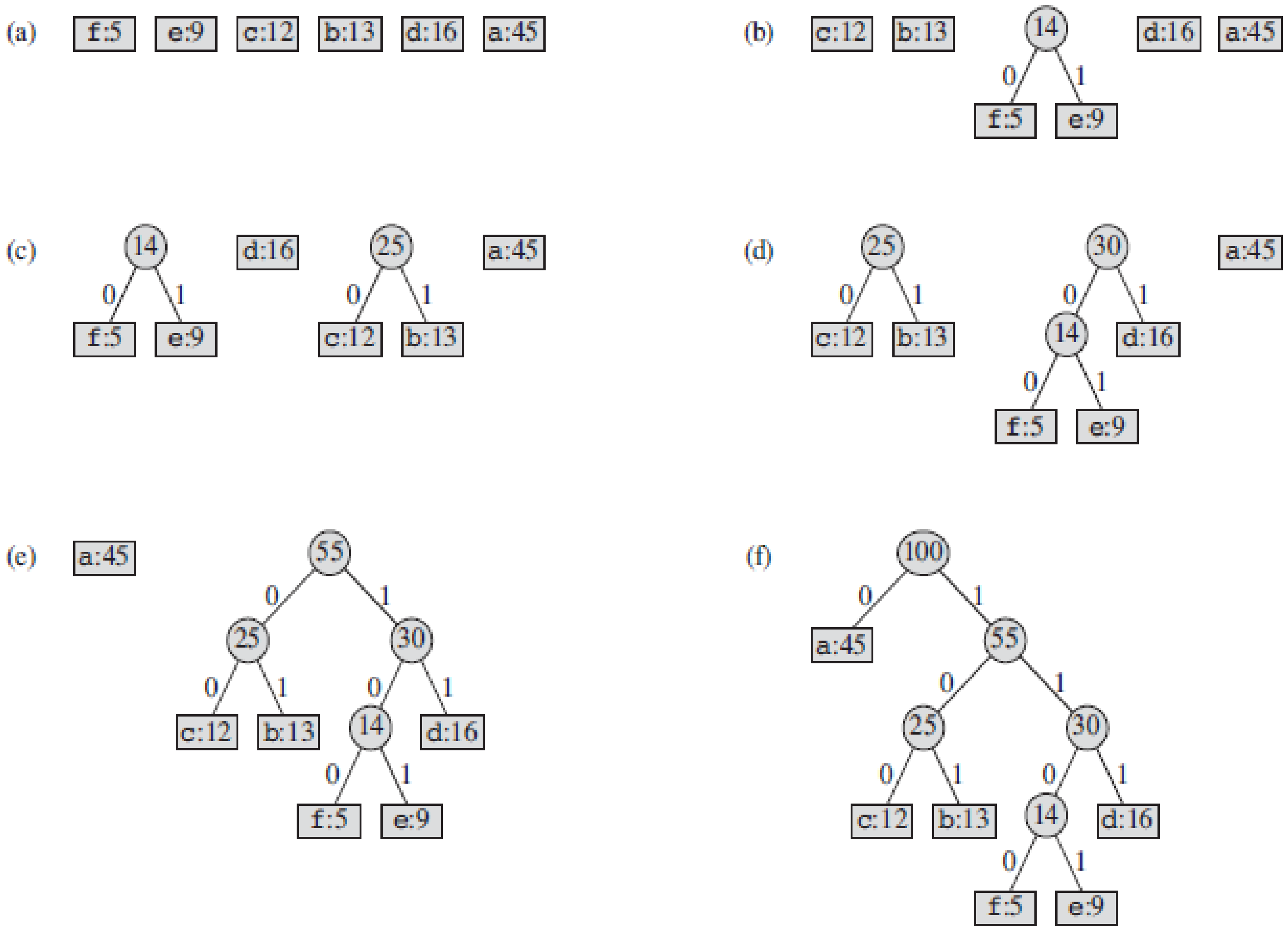
Constructing a Huffman Code

- Huffman developed a greedy algorithm for constructing an optimal prefix code
  - The algorithm builds the tree in a bottom-up manner
  - It begins with the leaves, then performs merging operations to build up the tree
  - At each step, it merges the two least frequent members together
    - It removes these characters from the set, and replaces them with a “metacharacter” with frequency = sum of the removed characters’ frequencies

Algorithm

```
HUFFMAN(C)
n = |C|
Q = C
for i = 1 to n - 1
    allocate a new node z
    z.left = x = EXTRACT-MIN(Q)
    z.right = y = EXTRACT-MIN(Q)
    z.freq = x.freq + y.freq
    INSERT(Q, z)
return EXTRACT-MIN(Q)    // return the root of the tree
```

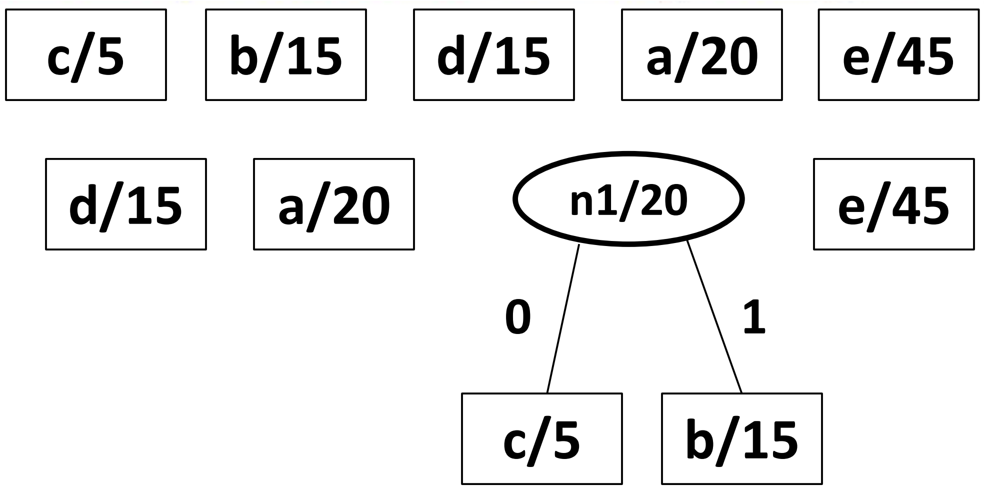
Example


(a)
f:5
e:9
c:12
b:13
d:16
a:45
(b)
c:12
b:13
14
0
1
f:5
e:9
d:16
a:45
(c)
14
0
1
f:5
e:9
d:16
25
0
1
c:12
b:13
a:45
(d)
25
0
1
c:12
b:13
30
0
1
14
d:16
0
1
f:5
e:9
a:45
(e)
a:45
55
0
1
25
30
0
1
c:12
b:13
0
1
14
d:16
0
1
f:5
e:9
(f)
100
0
1
a:45
55
0
1
25
30
0
1
c:12
b:13
0
1
14
d:16
0
1
f:5
e:9

Example

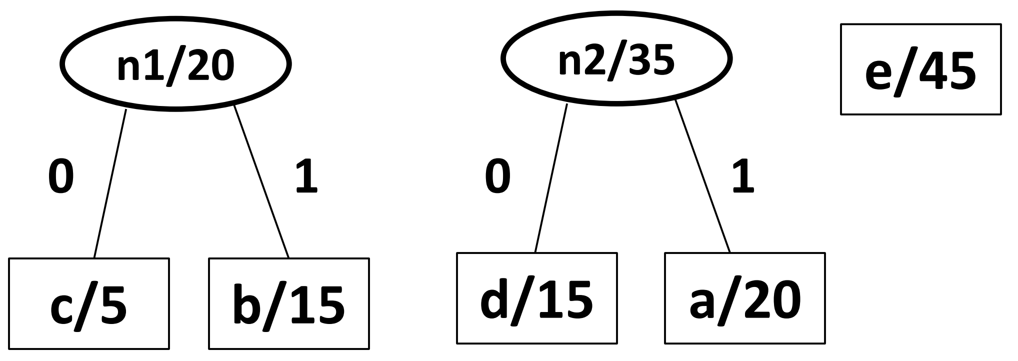
Let, $A = \{a/20, b/15, c/5, d/15, e/45\}$ be the alphabet(character) and its frequency distribution.

In the first step Huffman coding merges ***c*** and ***b***.



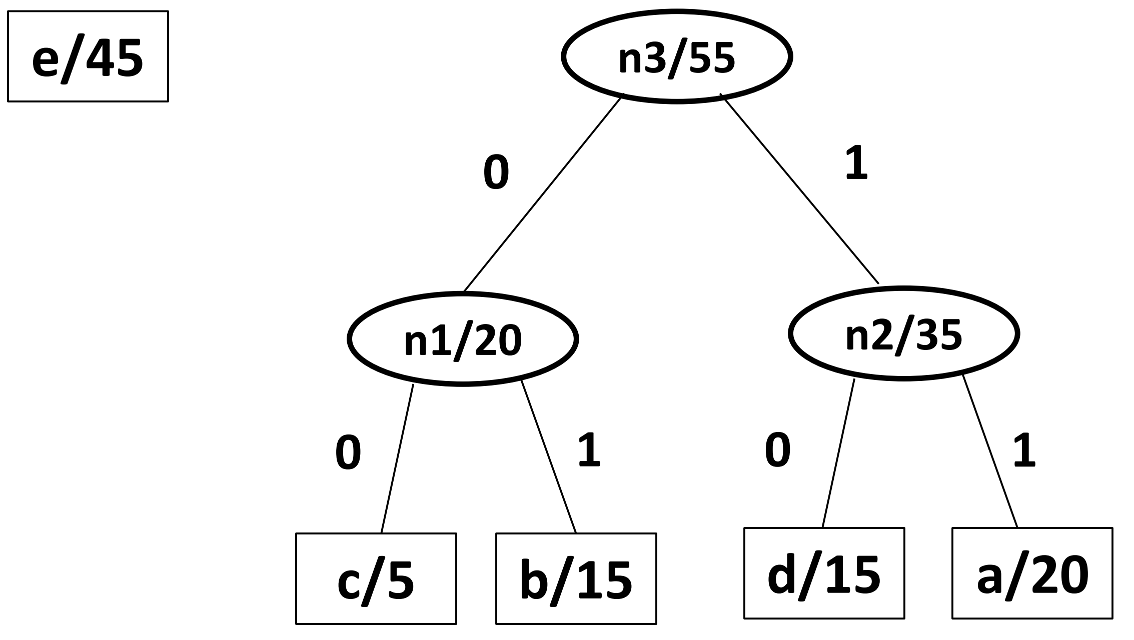
Alphabet is now $A_1 = \{a/20, b/15, n1/20, e/45\}$.

Alphabet is now $A_1 = \{a/20, b/15, n1/20, e/45\}$.
Algorithm now merges ***a*** and ***d***.
(could also have merged ***n1*** and ***d***).



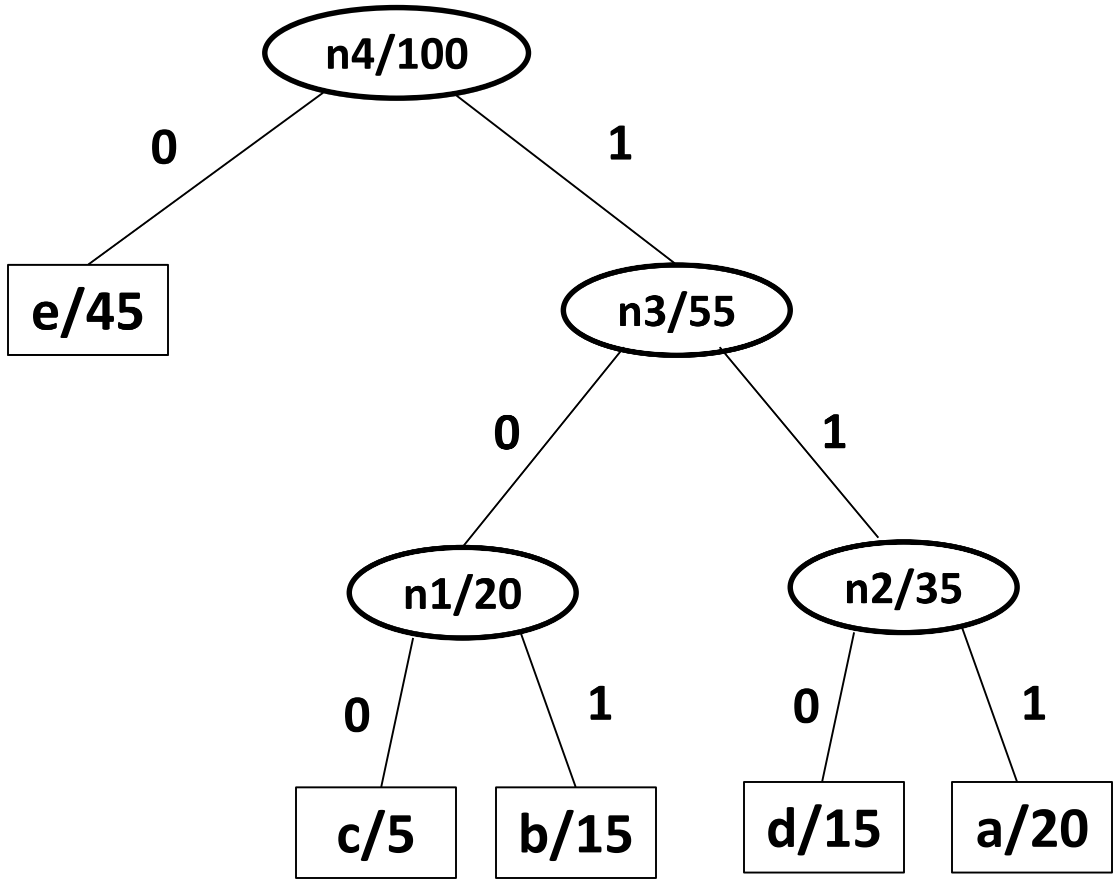
Alphabet is now $A_2 = \{n1/20, n2/35, e/45\}$.

Alphabet is $A_2 = \{n1/20, n2/35, e/45\}$.
Algorithm now merges ***n1*** and ***n2***.

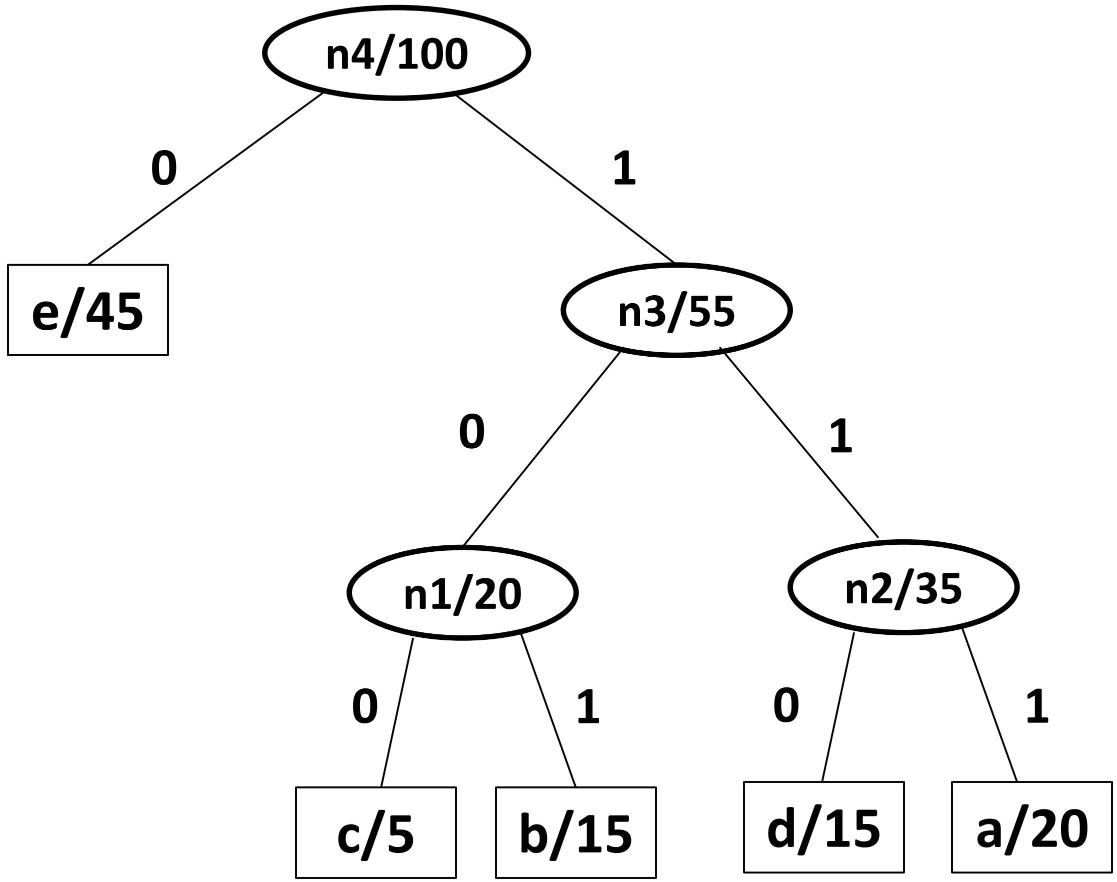


Now Alphabet is $A_3 = \{e/45, n3/55\}$.

Current Alphabet is $A_3 \; = \{e/45, n3/55\}$.
Algorithm now merges ***e*** and ***n3*** and finishes.



Huffman Code is obtained from the Huffman tree.



Huffman Code is
$a = 111, b = 101, c = 100, d = 110, e = 0.$
This is the optimum(minimum-cost) prefix code for this distribution.

- For example, we code the 3 letter file ‘abc’ as 111.101.100

Analysis

To analyze the running time of Huffman's algorithm, we assume that Q is implemented as a binary min-heap .For a set C of n characters, we can initialize Q in line 2 in O(n) time using the BUILD-MIN HEAP procedure. The **for** loop in lines 3–8 executes exactly (n -1) times, and since each heap operation requires time O(lg n), the loop contributes O(n lg n) to the running time.

Thus, the total running time of HUFFMAN on a set of n characters is O(n lg n).


Thank You!