Binary String Encoding

- Each character has a unique binary encoding, or codeword
- Fixed-length encoding
  - Each codeword has the same number of bits
  - Need $\lceil \log_2(n) \rceil$ bits to represent n distinct characters
  - Easy to identify individual characters within a string since characters are grouped by an equal # of bits

Representing a through f

a	000
b	001
c	010
d	011
e	100
f	101

110 and 111 are unused, but 2 bits is not enough

ASCII uses 8 bits

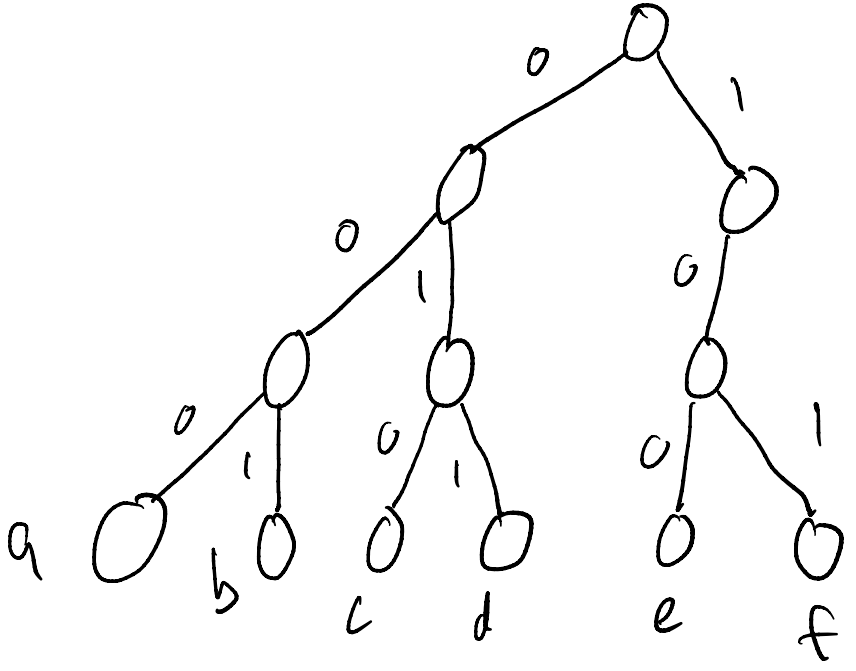
- Variable-length encoding
  - Some characters use more bits than others
  - Can save space if frequently used characters use fewer bits
  - In order to determine where one char ends and the next begins, no character's encoding may be the prefix of another character's encoding

    10 for a and 101 for b won't work

- To access an individual character at an arbitrary position, the entire string up to that position must be scanned

Prefix Tree (or trie)

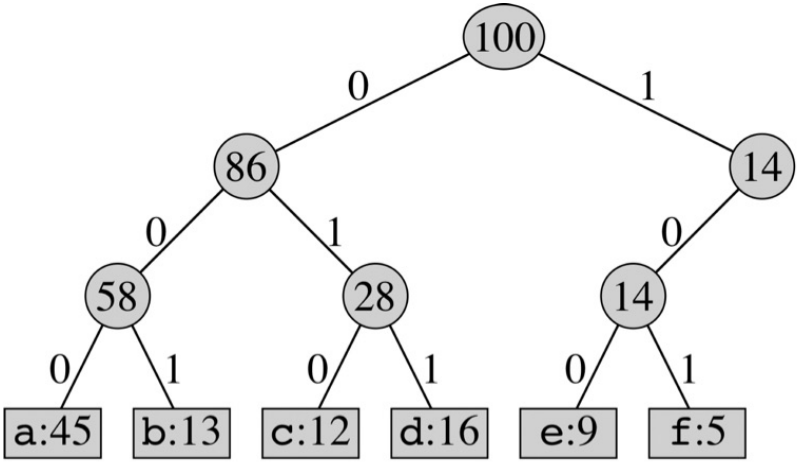
- Can be used to represent binary encodings
- The leaves are characters
- The path to a leaf represents the binary encoding of that character
- The edge to a left child is 0, the right is 1
- Since characters are leaves, no character's encoding is the prefix of another character's encoding



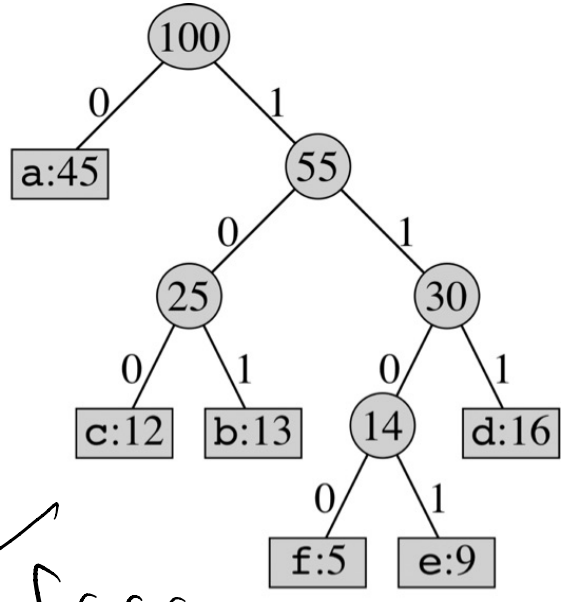
a b c

000 001 010

a b c



(a)



(b)

a b c

0 101 100

a b c

f a a a

1100000

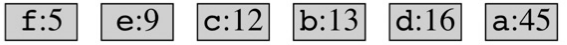
	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

Huffman Codes

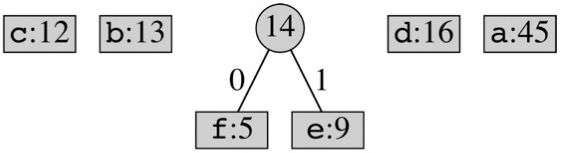
- Compresses strings by using a variable-length encoding based on character frequency
- Build a prefix tree where the paths from the root to the leaves for frequent characters are shorter than those for infrequent characters

- Building the tree
  - Each node stores a frequency
    - Leaves store a frequency and a character
    - An internal node stores the sum of the frequencies of the leaves of the subtree rooted at that node
  - Build the tree from the bottom up with repeated merging operations

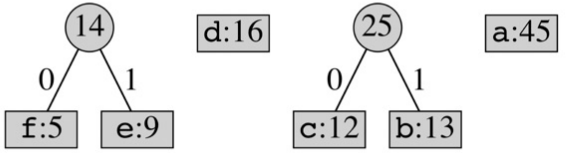
(a)


f:5
e:9
c:12
b:13
d:16
a:45

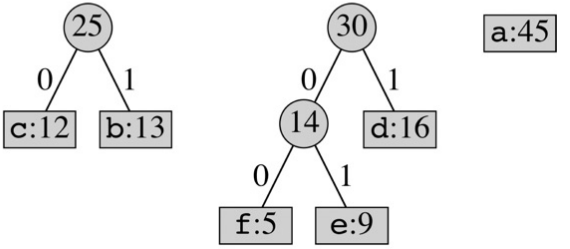
(b)


c:12
b:13
14
0
1
f:5
e:9
d:16
a:45

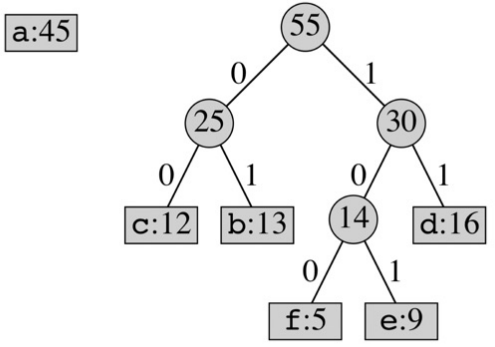
(c)


14
0
1
f:5
e:9
d:16
25
0
1
c:12
b:13
a:45

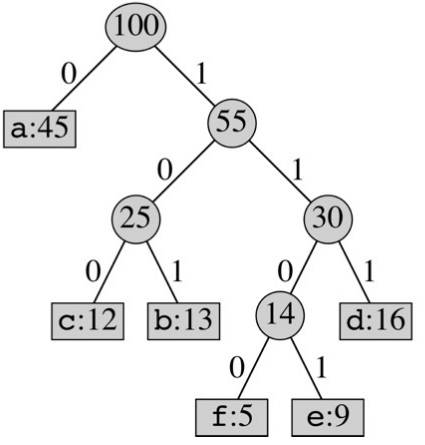
(d)


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
0
1
c:12
b:13
14
d:16
0
1
f:5
e:9

(f)


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

C is a set containing the characters and their frequencies as objects

```
HUFFMAN(C)
1  n = |C|
2  Q = C        Q is a min-heap priority queue built from C   Θ(n)
3  for i = 1 to n − 1
4      allocate a new node z
5      z.left = x = EXTRACT-MIN(Q)                            Θ(lg n)
6      z.right = y = EXTRACT-MIN(Q)                           Θ(lg n)
7      z.freq = x.freq + y.freq
8      INSERT(Q, z)                                           Θ(lg n)
9  return EXTRACT-MIN(Q)      // return the root of the tree
```

Θ(lg n)

whole thing is Θ(n lg n)

Greedy Algorithms

- Choose a locally optimal choice in the hope that that it will lead to a globally optimal solution
    - Make whatever choice seems best in the moment
- For some problems, a greedy solution leads to the optimal solution
    - Building Huffman codes, always choose the remaining nodes with the lowest frequencies to merge

- Greedy algorithms usually start out by sorting or building a priority queue for efficient selection of the next "best" item