BOOK

A Simplified Approach
to
Data Structures

Prof.(Dr.)Vishal Goyal, Professor, Punjabi University Patiala
Dr. Lalit Goyal, Associate Professor, DAV College, Jalandhar
Mr. Pawan Kumar, Assistant Professor, DAV College, Bhatinda

Shroff Publications and Distributors
Edition 2014

EXPRESSION TREE
&
HUFFMAN ALGORITHM

Department of Computer Science, Punjabi University Patiala

CONTENTS FOR TODAY’S LECTURE

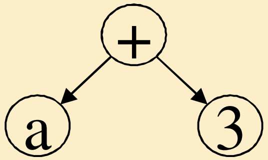
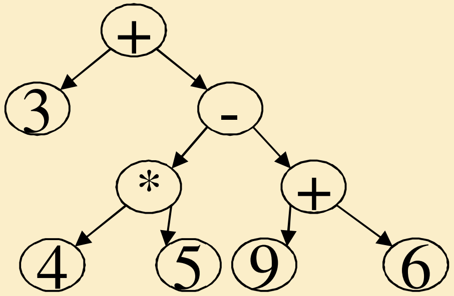
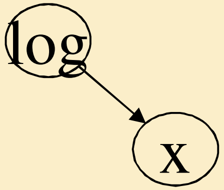
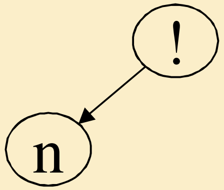
- Introduction to Expression trees
- Why Expression Tree
- Implementation of Expression trees
- Introduction to Huffman Algorithm
- Prefix Codes
- Huffman Code : Construction
- Huffman Code : Decoding

EXPRESSION TREES

- An expression tree for an arithmetic, relational, or logical expression is a binary tree in which :

- The parentheses in the expression do not appear.
- The leaves are the variables or constants in the expression.
- The non-leaf nodes are the operators in the expression :
  - A node for a binary operator has two non-empty subtrees.
  - A node for a unary operator has one non-empty subtree.

Example of Expression Tree

Expression	Expression Tree	Inorder Traversal Result
(a+3)		a + 3
3+(4*5-(9+6))		3+4*5-9+6
log(x)		log x
n!		n !

Why Expression Trees?

- Expression trees are used to remove ambiguity in expressions.
- Consider the algebraic expression 2 - 3 * 4 + 5.
- Without the use of precedence rules or parentheses, different orders of evaluation are possible :

((2-3)*(4+5)) = -9
((2-(3*4))+5) = -5
(2-((3*4)+5)) = -15
(((2-3)*4)+5) = 1
(2-(3*(4+5))) = -25

- The expression is ambiguous because it uses infix notation : each operator is placed between its operands.

Why Expression trees? (contd.)

- Storing a fully parenthesized expression, such as ((x+2)-(y*(4-z))), is wasteful, since the parentheses in the expression need to be stored to properly evaluate the expression.
- A compiler will read an expression in a language like Java, and transform it into an expression tree.
- Expression trees impose a hierarchy on the operations in the expression. Terms deeper in the tree get evaluated first. This allows the establishment of the correct precedence of operations without using parentheses.
- Expression trees can be very useful for:
  - Evaluation of the expression.
  - Generating correct compiler code to actually compute the expression's value at execution time.
  - Performing symbolic mathematical operations (such as differentiation) on the expression.

Implementing the Expression Tree

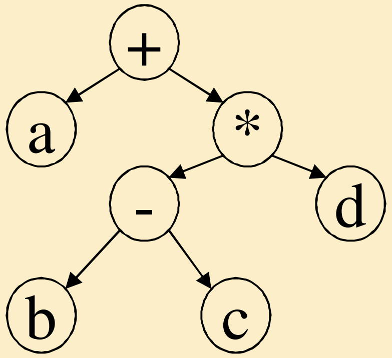
Expression Trees can be achieved by using three notations. These are :

- Prefix Notation
- Infix Notation
- Postfix Notation

Prefix Notation

- A preorder traversal of an expression tree yields the prefix (or polish) form of the expression.
- In this form, every operator appears before its operand(s).

For Example , Consider the tree :

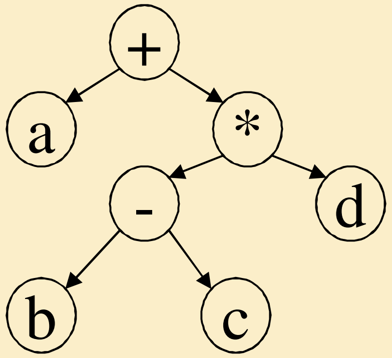


Prefix Notation : + a * - b c d

Infix Notation

- An inorder traversal of an expression tree yields the infix form of the expression.
- In this form, every operator appears between its operand(s).

For Example , Consider the tree :

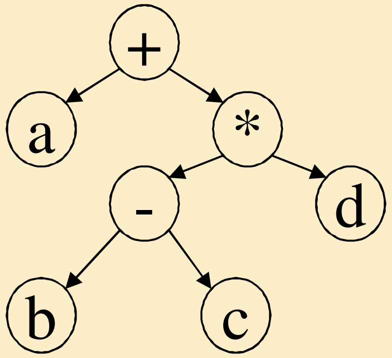


Infix Notation : a + b - c * d

Postfix Notation

- An postorder traversal of an expression tree yields the postfix form of the expression.
- In this form, every operator appears after its operand(s).

For Example , Consider the tree :



Postfix Notation : a b c - d * +

Prefix, Infix, and Postfix Forms (contd.)

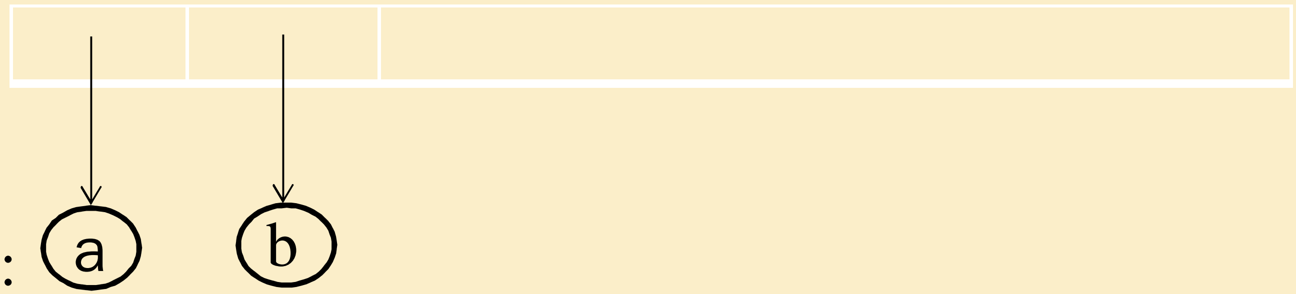
Expression	Prefix forms	Infix forms	Postfix forms
(a + b)	+ a b	a + b	a b +
a - (b * c)	- a * b c	a - b * c	a b c * -
log (x)	log x	log x	x log
n !	! n	n !	n !

Expression Tree from Postfix Notation

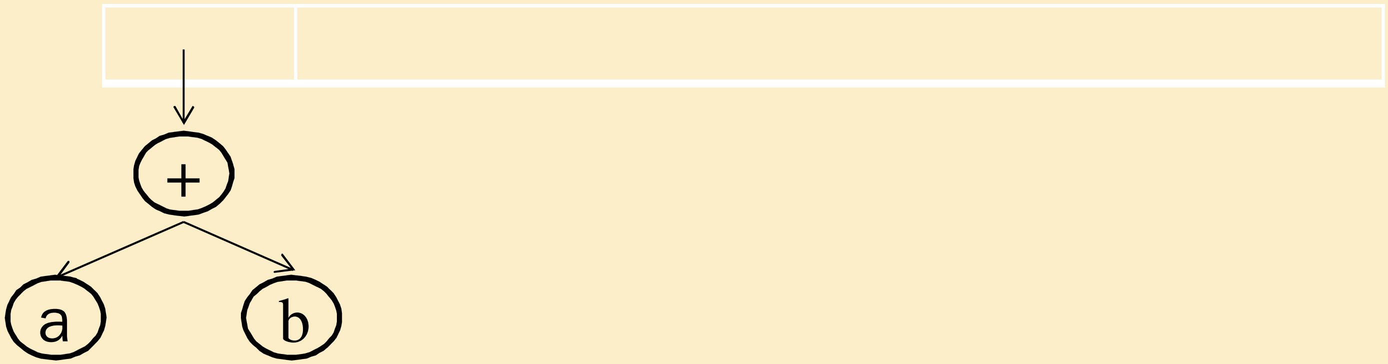
Consider the expression (a + b) * c. The postfix expression is:

a b + c *

Step 1 :

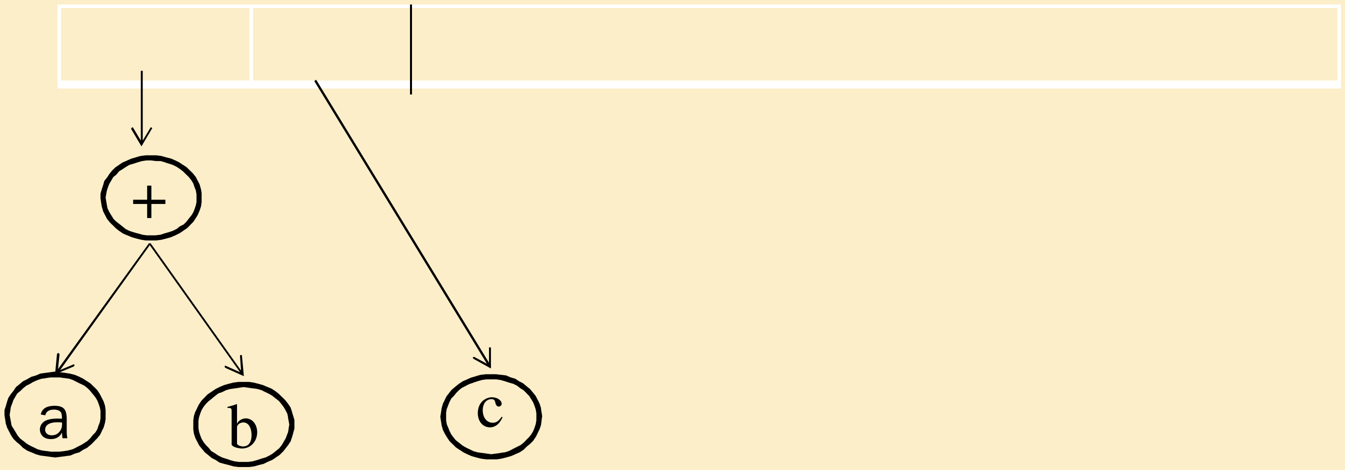


Step 2 :

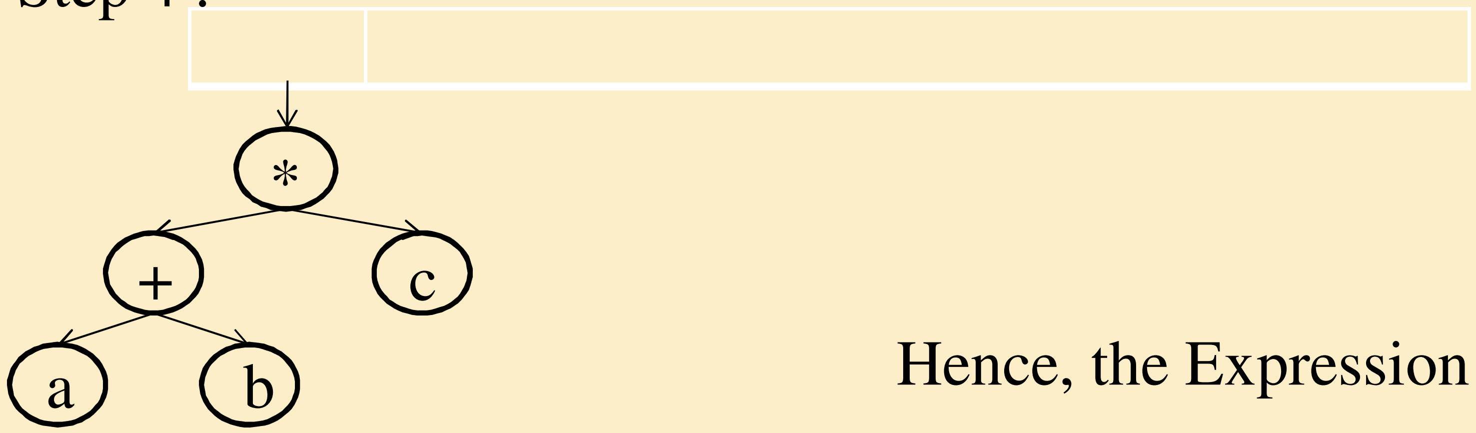


Expression Tree from Postfix Notation

Step 3 :



Step 4 :



Hence, the Expression Tree.

NOTE : For a computer generator program constructing an expression tree from infix notation is not preferred. Instead , a computer program uses postfix expression to express the expression tree. Because in postfix expression there is no need to apply rules of precedence and associativity.

HUFFMAN ALGORITHM

MOTIVATION

- Suppose we want to store and transmit very large files (messages) consisting of strings (words) constructed over an alphabet of characters (letters).
- Representing each character with a fixed-length code will not result in the shortest possible file!
- Example: 8-bit ASCII code for characters

– some characters are much more frequent than others

– using shorter codes for frequent characters and longer ones for infrequent ones will result in a shorter file.

Coding : Problem Definition

- Represent the characters from an input alphabet using a variable-length code alphabet C, taking into account the occurrence frequency of the characters.
- Desired properties:
  - The code must be uniquely decipherable: every message can be decoded in only one way.
  - The code must be optimal with respect to the input probability distribution.
  - No string is a prefix of another.

Example

Character	a	b	c	d	e	f
Frequency (%)	45	13	12	16	9	5
Fixed Length	000	001	010	011	100	101
Variable Length	0 101	101	100	111	1101	1100

Message: abadef

Fixed Length : 000001000011100101

Variable Length : 0101011111011100

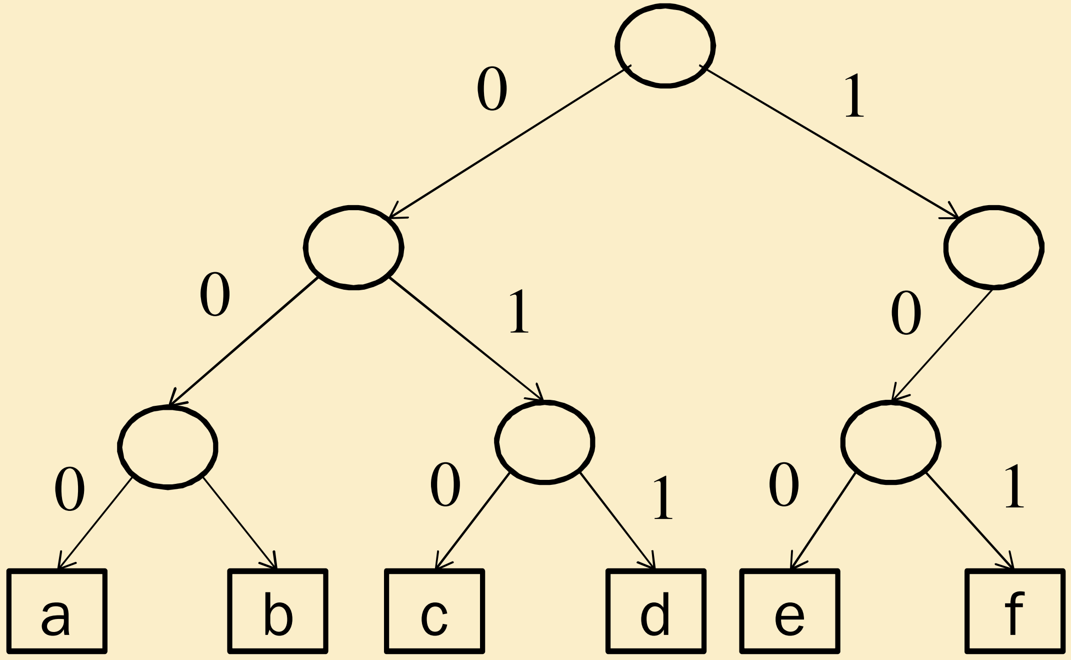
A file of 100,000 characters takes:

- 3×100,000 = 300,000 bits with fixed-length code
- (.45×1 + .13×3 + .12×3+ .16×3 + .09×4 + .05×4) ×100,000 = 224,000 bits on average with variable-length code (25% less)

Prefix Codes

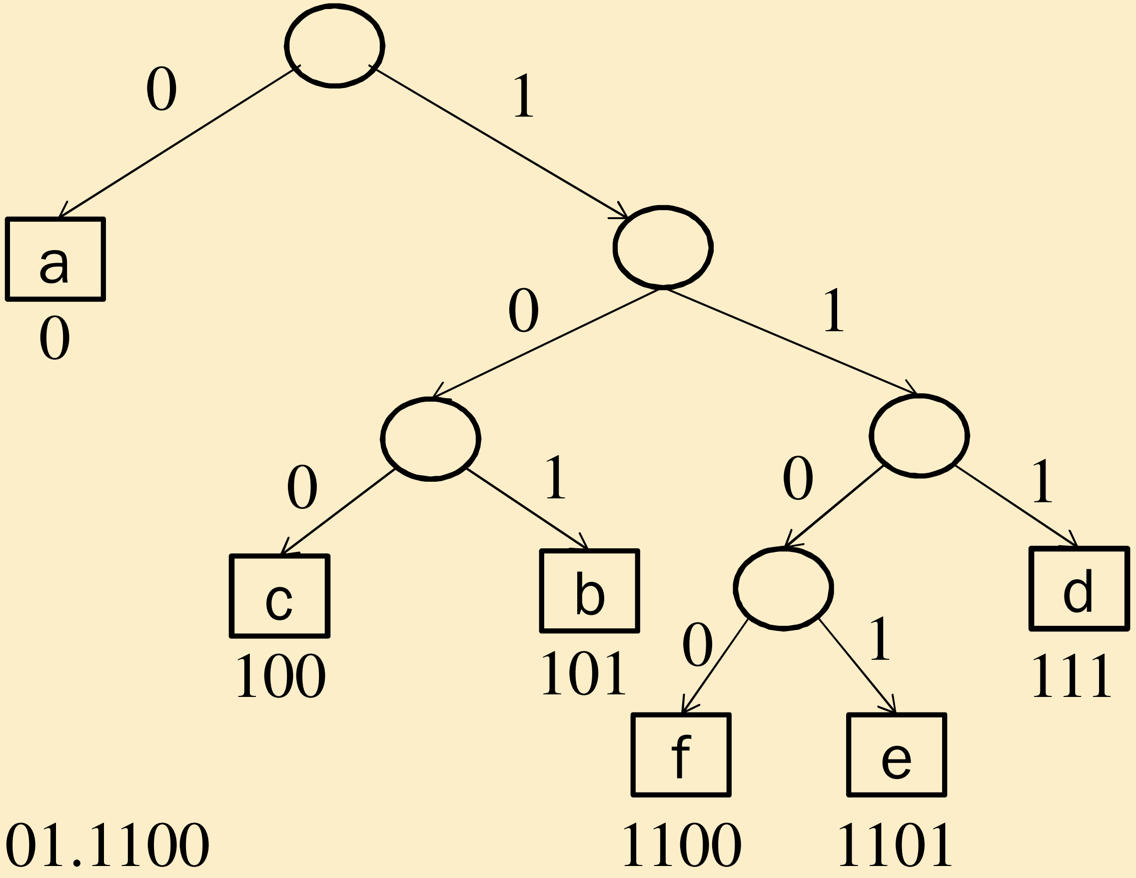
- We consider only prefix codes: no code-word is a prefix of another code-word. Prefix codes are uniquely decipherable by definition.
- A binary prefix code can be represented as a binary tree:
  - leaves are a code-words and their frequency (%)
  - internal nodes are binary decision points: “0” means go to the left, “1” means go to the right of a character. They include the sum of frequencies of their children.
  - The path from the root to the code-word is the binary representation of the code-word.

Example: fixed-length prefix code (1)



Message: 000.001.000.011.100.101 abadef

Example: variable-length prefix code (2)



Message: 0.101.0.111.1101.1100

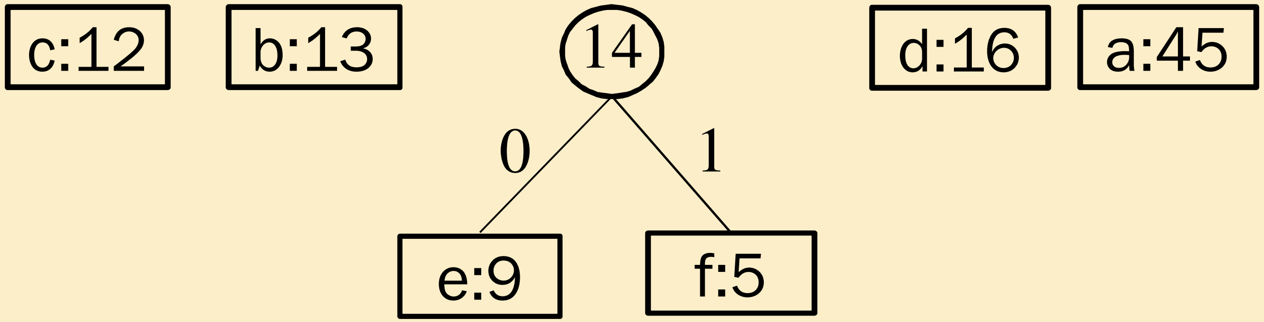
Huffman code: construction

- Idea: build the tree bottom-up, starting with the code-words as leafs of the tree and creating intermediate nodes by merging the new object whose frequency is the sum of the frequencies of the merged objects.
- To efficiently find the two least-frequent objects, use a minimum priority queue.
- The result of the merger of two objects is a new object whose frequency is the sum of the frequencies of the merged objects.

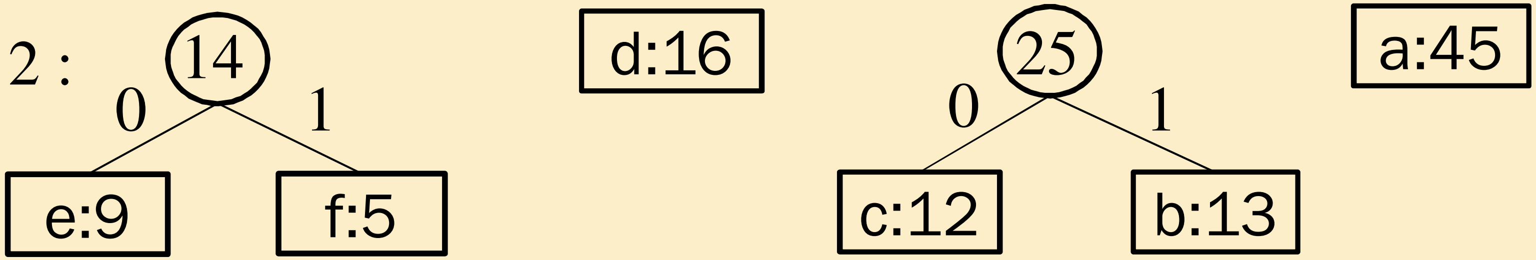
Example : Huffman Code Construction(1)

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

Step 1 :

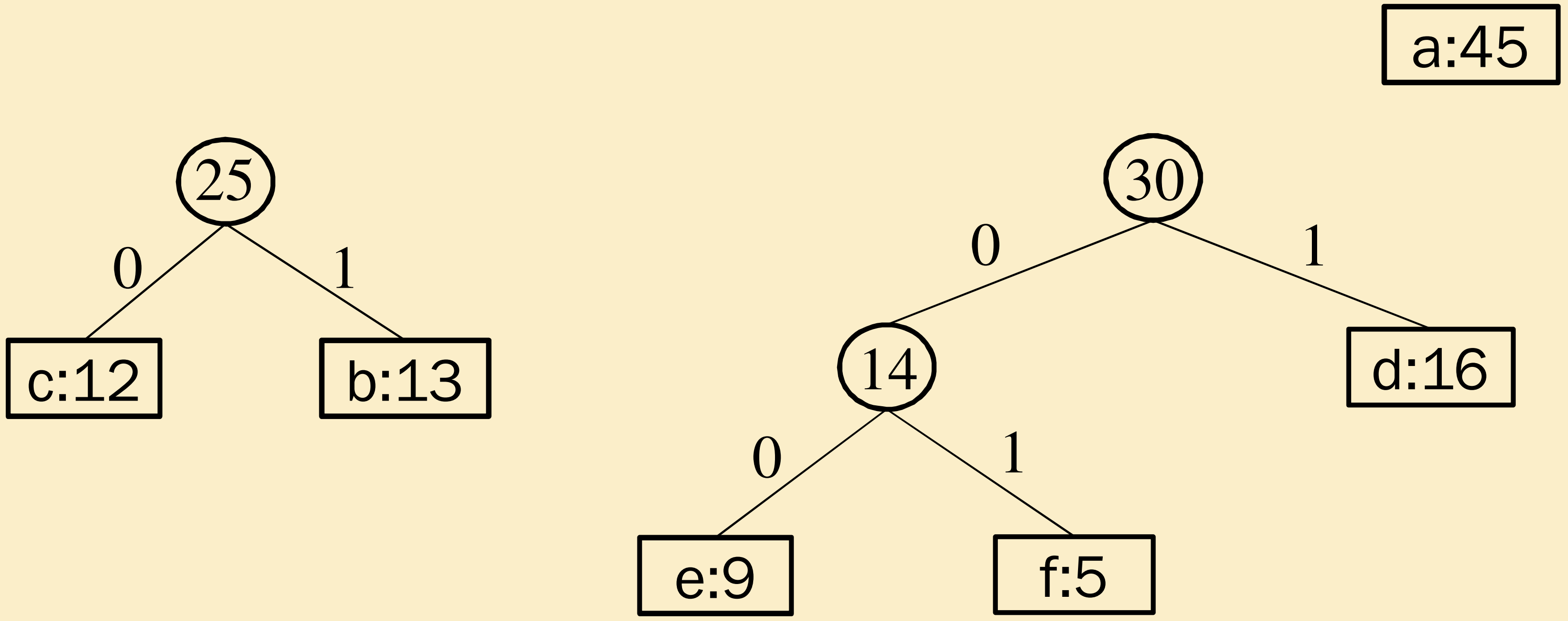


Step 2 :



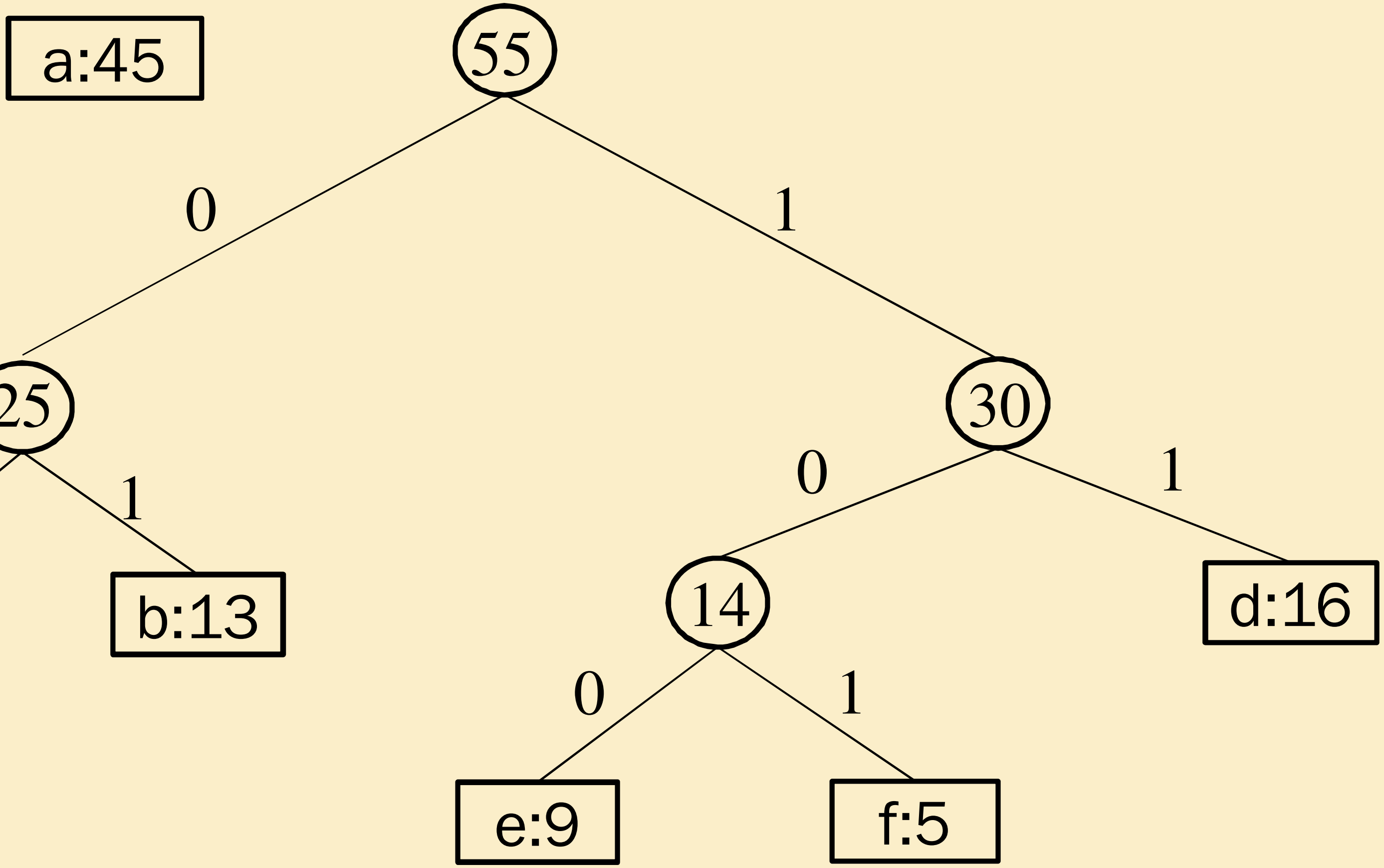
Example : Huffman Code Construction(2)

Step 3:



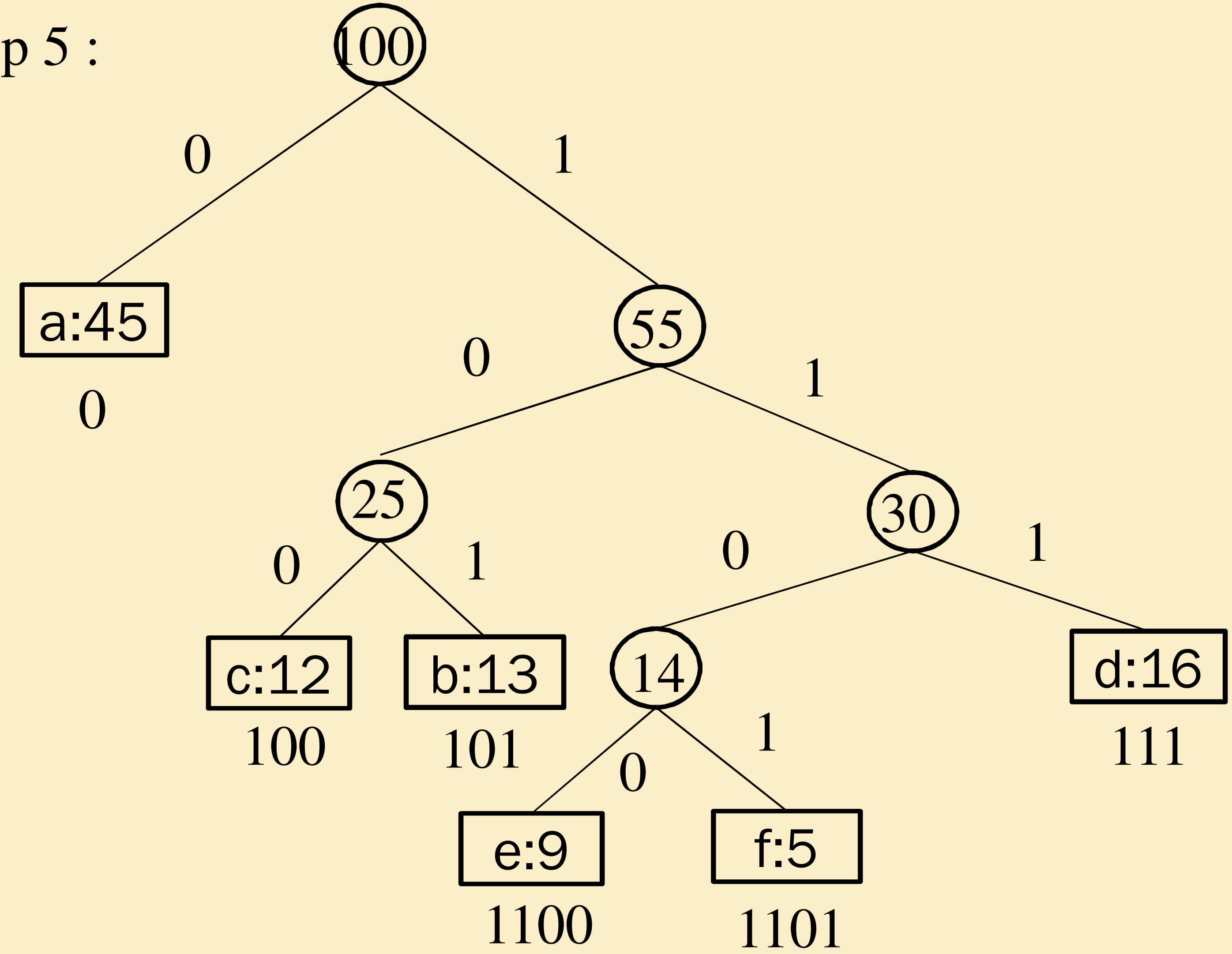
Example : Huffman Code Construction(3)

Step 4 :



Example : Huffman Code Construction(4)

Step 5 :



Example : Huffman Code Construction(5)

Result : Codes for the variables :-

a : 0

b : 100

c : 101

d : 111

e : 1100

f : 1101

Hence, no code is the prefix of another code.

Huffman code: decoding

- Huffman invented in 1952 a greedy algorithm for constructing an optimal prefix code, called a Huffman code.
- Decoding:
    1. Start at the root of the coding tree T, read input bits.
    2. After reading "0" go left
    3. After reading "1" go right
    4. If a leaf node has been reached, output the character stored in the leaf, and return to the root of the tree.

Complexity: O(n), where n is the message length