Application of Trees

Application: Expression Trees

- Arithmetic expressions:
  - Arithmetic infix-expression: binary operator appears between it's two operands
    For example, $5\ +\ 6\ -\ 4\ *\ 3\ /\ 2\ -\ 1\ *\ 7\ +\ 4$
  - Arithmetic prefix-expression: binary operator appears right before it's two operands
    For example, $+\ -\ -\ +\ 5\ \ 6\ /\ *\ 4\ \ 3\ \ 2\ *\ 1\ \ 7\ \ 4$
  - Arithmetic postfix-expression: binary operator appears right after it's two operands
    For example, $5\ \ 6\ +\ 4\ \ 3\ *\ 2\ /\ -\ \ 1\ \ 7\ *\ -\ 4\ +$
- Why prefix and postfix expressions?

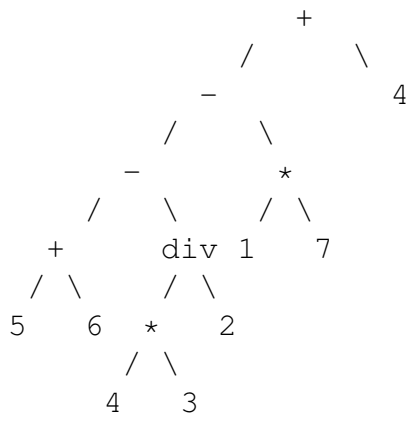
  Because arithmetic postfix expressions can be easily evaluated by a computer algorithm using a stack. Given a postfix expression, scan through the expression token by token.
  - If next token read is an operand, push it to the stack
  - If next token is an operator, pop two operands from the stack and apply the operator to these two operands, and then push back the result to the stack.
  - At the end, if the stack contains a single value, it means the expression is a valid expression and the value is the evaluation result of the expression. Otherwise, the expression is invalid.

  Give examples in class.
- Converting from infix to prefix and postfix expressions: human algorithm

  Given an infix expression such as $5\ +\ 6\ -\ 4\ *\ 3\ /\ 2\ -\ 1\ *\ 7\ +\ 4$, we can
  - fully parenthesize the expression to $((((5+6)-((4*3)/2))-(1*7))+4)$
  - Converting to the prefix expression:
    (1) For each operator, replace the corresponding opening parenthesis by the operator;
    (2) Remove all the closing parentheses.
  - Converting to the postfix expression:
    (1) For each operator, replace the corresponding closing parenthesis by the operator;
    (2) Remove all the opening parentheses.
- Converting from infix to prefix and postfix expressions: computer algorithm
  Given an infix expression such as $5\ +\ 6\ -\ 4\ *\ 3\ /\ 2\ -\ 1\ *\ 7\ +\ 4$, we can

– convert the arithmetic expression to an expression tree, where each internal node represents an operator with its left subtree representing the left operand and its right subtree representing the right operand.

For example, the expression tree for the given expression is



– The arithmetic prefix expression can be formed by performing a pre-order traversal on the expression tree.

– Similarly, the arithmetic postfix expression can be formed by performing a post-order traversal on the expression tree.

Application: File Compressions using Huffman Code

- Purpose: data compression/encoding.

  In a ASCII text file, each char requires a byte (8 bits).

  To reduce the size of the file, we can encode each char by a variable code. For example:

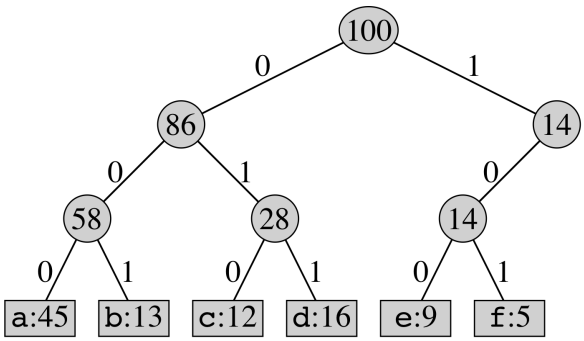
	a	b	c	d	e	f
Frequency (in thousands)	45	13	12	16	9	5
Fixed-length codeword	000	001	010	011	100	101
Variable-length codeword	0	101	100	111	1101	1100

  Fixed-length codes: 3-bit code x 100000 = 300,000 bits.

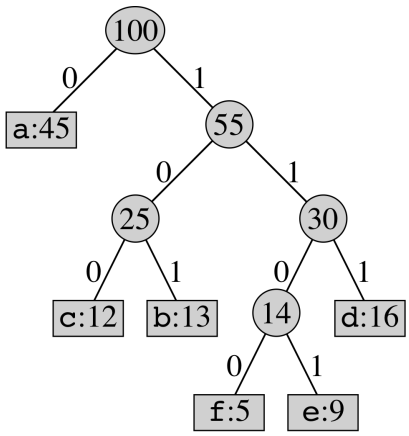
  Variable length codes: $(45x1 + 13*3 + 12*3 + 16*3 + 9*4 + 5*4)*1000 = 224,000$ bits

- Depending on the frequencies of all characters appearing in a document, we can use less then 8-bit to represent the chars with higher frequencies and use more than 8-bit for those less-frequent chars. As a result, the overall size of the encoded file will be smaller.

  We consider **prefix codes** in which no codeword is also a prefix of some other codes. Encoding and decoding are both simple using prefix codes. The actual code, known as **Huffman Code** is represented by a tree.



(a)



(b)

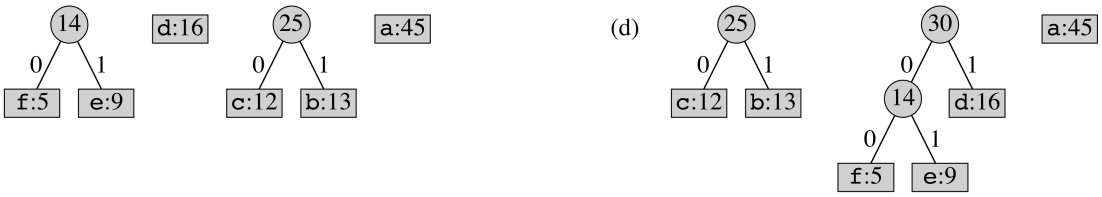
- Huffman encoding process:
  - Huffman tree construction:
    1. All the characters are in leaf nodes. For each internal node, the left branch means 0 and the right branch means 1. The Huffman code for each character is the 0-1 bit-string represented by the path from the root to the corresponding leaf node.
    2. The location of each character depends on the expected frequency of that character appearing in a document.
    3. At beginning, we can construct a Huffman tree from a given text file by counting the occurrences of all characters in the file. Let's call it a Huffman Tree Construction File (HTCF).
    4. HTCF needs to be shared by two communicating parties so that they can construct the same Huffman Tree for successful encoding and decoding.

    Tree Construction algorithm:

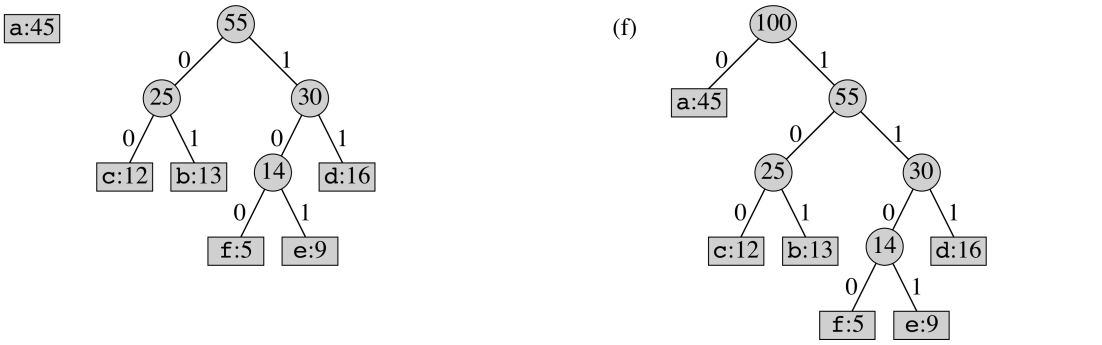
    1. Count the frequencies of all characters in a given file (HTCF).
    2. Construct a priority queue containing a set of Huffman trees, where smaller weight of the tree (sum of frequency counts of all chars in the tree) has a higher priority
    3. At beginning, the priority queue contains all single-node Huffman trees, where each Huffman tree contains only one character.
    4. Extract two Minimum Huffman Trees from the priority queue
    5. Combine these two Huffman trees into one tree by creating a new root and these two Huffman trees will be left and right subtrees of the new root
    6. Insert the combined Huffman tree back to the priority queue
    7. Repeat steps 4, 5 and 6 until only one Huffman tree left in the priority queue

(a) (b)

(c) (d)



(e) (f)



– File encoding:

  Based on the constructed Huffman tree, we can create a mapping table of characters and their bit-strings.

– File decoding:

  Upon receiving an encoded file, the receiving end can based on the same mapping table to decode the file.