

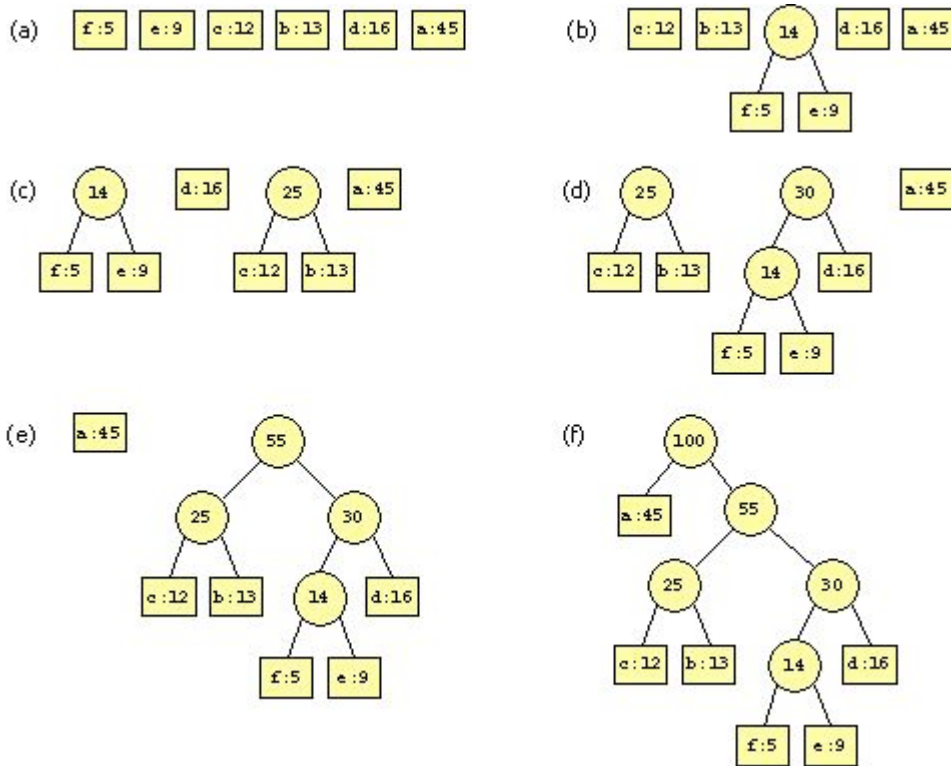
Study Project: Data Compression

In 1952 David A. Huffman published the paper "A Method for the Construction of Minimum-Redundancy Codes" in which he defined a greedy algorithm that constructs an optimal prefix code called a **Huffman code**.

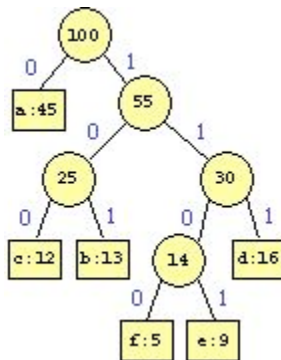
The algorithm begins with a set C of pairs (c, f) , called `HuffmanLeaf`, where c is a character of an underlying alphabet (e.g., ASCII or data type `byte`) and f denotes the number of occurrences of c in a given source file. The set C is a partial order \leq over pairs. Thus, if $(\text{'t'}, 2), (\text{'d'}, 3) \in C$ (i.e., $(\text{'t'}, 2), (\text{'d'}, 3)$ are elements of C), then we have that pair $(\text{'t'}, 2) \leq (\text{'d'}, 3)$. Huffman's algorithm uses the frequency f to identify the two least-frequent objects in C and merges them. The result of the merger is a new object (i.e., `HuffmanNode`) whose frequency is the sum of the frequency of the two objects being merged. In total, the algorithm performs a sequence of $|C| - 1$ "merging" operations to create a Huffman tree.

How does it work?

Assume a file that contains 100 characters built out of 6 different letters with the following frequency: 'a' : 45, 'b' : 13, 'c' : 12, 'd' : 16, 'e' : 9, and 'f' : 5. The Huffman algorithm creates a `HuffmanTree` as follows:



How we need to construct the Huffman codes. An edge connecting an internal node with its children is labeled "0" if it is an edge to the left child, and "1" if it is an edge to the right child. The Huffman code for a character c is the sequence of labels on the edges connecting the root to the leaf for that character.



Therefore, we encode:

```

`a`: 0
`b`: 101
`c`: 100
`d`: 111
`e`: 1101
`f`: 1100
  
```

The compression ratio can be calculated as follows. We start with the ASCII encoding. Every character in the encoding requires 8 bits. Thus, a text containing 100 characters has a size of 800 bits, or 100 Byte. The number of bits required using the calculated Huffman codes is $45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4 = 45 + 39 + 36 + 48 + 36 + 20 = 244$, 28 Byte, which yields a compression ratio of 72%.

Savings of 20% to 90% are typical, but not guaranteed. In fact, Huffman compression is less effective than Lempel-Ziv compression.

Compression: Design idea

Huffman codes are constructed in two phases: (i) an analysis phase in which the whole source file is read to construct a `map<unsigned char, long>` of pairs `(Character, Frequency)`, and (ii) a coding phase in which a Huffman tree is constructed to determine the Huffman codes.

In order to implement Huffman compression you need to solve the following sub-problems:

1. Frequency collection, build the set `C`:

Start with an empty `map<unsigned char, long>` `frequencies`. You need to process the whole input file, read all characters, and count their occurrences, that is, `frequencies[c]++`. Start with an empty `map<unsigned char, long>`.

2. Huffman tree construction:

Construct a Huffman tree according to the Huffman algorithm. The best data type for this purpose is `set<T>`, where `T` is the base type of the set. `T` should be a super type (or interface) for Huffman leaves and Huffman nodes. Also, to use the set, you need to define a proper `<` operator for Huffman leaves and nodes.

3. Huffman code assignment:

Annotate the edges of the Huffman tree with 0 or 1. That is, define a scheme to assign every leaf-node its corresponding Huffman code.

4. Huffman character/code mapping:

Build a `map<unsigned byte, HuffmanCode>` that contains mappings from characters to `HuffmanCode`'s. `HuffmanCode` is the data type you chose to represent Huffman codes. You need to traverse the Huffman tree and visit all leafs. Use object-oriented techniques to solve this problem.

Assume as source files plain binary files. That is, a character is represented as a `unsigned char`. Huffman codes have flexible length. To facilitate their representation it might be useful, but not required, to use the data type `string`. For example, the character `'e'` is represented as `0x65` of type `unsigned char` and its corresponding Huffman code as `"1101"` of type `string`.

Your solution will require several different classes. For example, you will need at least, a class for the analysis phase, the class `HuffmanLeaf` to represent leaves, the class `HuffmanNode` to represent nodes, and the class `HuffmanTree` to represent Huffman trees.

To test your application you can use the file `"sample.txt"`, which contains the text from problem set 2. This file is 1996 Bytes long and contains 53 different characters. The expected compression ratio is 41.27%, that is, we can represent 15968 Bits (1996 Bytes) of fixed-size code by at most 9384 Bits (or 1173 Bytes) of variable-size prefix codes.

Week 1

Study the Huffman algorithm. Search the Internet for additional information. Develop a basic class structure. Familiarize yourself with the container types `map` and `set`. Start with simple tests. Prepare a catalogue of questions that need to be discussed in class.

Week 2

Implement the frequency analysis and the construction of the Huffman tree.

Week 3

Implement the compression.

Week 4

Implement the decompression.

Week 5

Code cleanup.

Deliverable

An operational compression/decompression C++ console application.

We will discuss ideas, problems, and possible solutions in class.

Lab sessions: 03/20, 03/22, and 03/27

Final submission deadline: Thursday, April 19, 2004, 4:10 p.m.