m-ary trees

Definition: A rooted tree is called an **m-ary tree** if every internal vertex has no more than m children.

The tree is called a **full m-ary tree** if every internal vertex has exactly m children.

An m-ary tree with m = 2 is called a **binary tree**.

Theorem 2: A tree with n vertices has (n – 1) edges.

Theorem 3: A full m-ary tree with i internal vertices contains n = mi + 1 vertices.

We did these theorems from page 752 (p. 690, 6th ed.) last time.

More m-ary trees

From Theorem 3: A full m-ary tree with i internal vertices contains n = mi + 1 vertices we immediately get:

Theorem 4 (p. 753; 691 6th ed.): A full m-ary tree with

1. n vertices has i = (n-1)/m internal vertices and l = ((m-1)n + 1)/m leaves.
2. i internal vertices has n = mi+1 vertices and l = (m-1)i + 1 leaves.
3. l leaves has n = (ml-1)/(m-1) vertices and i = (l-1)/(m-1) internal vertices.

This means that for a full m-ary tree any one of these numbers determines the other two.

Proof: from Theorem 3, n = mi + 1.

For 1, solve for i, i = (n-1)/m,
l =n–i= n – (n-1)/m = ((m-1)n+1)/m

For 2, Th.3 gives the first part, and
l = n-i=(mi+1)-i = (m-1)i +1

For 3, solve the formula for l in terms of n from part 1 for n in terms of l, then subtract to get the formula for i.

Huffman Coding Trees

We usually encode strings by assigning **fixed-length codes** to all characters in the alphabet (for example, 8-bit coding in ASCII).

However, if different characters occur with different frequencies, we can save memory and reduce transmittal time by using **variable-length encoding**.

The idea is to assign shorter codes to characters that occur more often.

Huffman Coding Trees

We must be careful when assigning variable-length codes.

For example, let us encode **e** with 0, **a** with 1, and **t** with 01. How can we then encode the word **tea**?

The encoding is **0101**.

Unfortunately, this encoding is ambiguous. It could also stand for **eat**, **eaea**, or **tt**.

Of course this coding is unacceptable, because it results in loss of information.

Huffman Coding Trees

To avoid such ambiguities, we can use **prefix codes**. In a prefix code, the bit string for a character never occurs as the **prefix** (first part) of the bit string for another character.

For example, the encoding of **e** with 0, **a** with 10, and **t** with 11 is a prefix code. How can we now encode the word **tea**?

The encoding is **11010**.

This bit string is unique, it can only encode the word tea.

Huffman Coding Trees

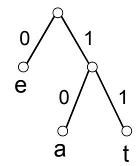
We can represent prefix codes using binary trees, where the characters are the labels of the leaves in the tree.

The edges of the tree are labeled so that an edge leading to a left child is assigned a 0 and an edge leading to a right child is assigned a 1.

The bit string used to encode a character is the sequence of labels of the edges in the unique path from the root to the leaf labeled with this character.

Huffman Coding Trees

The tree corresponding to our example:



In a tree, no leaf can be the ancestor of another leaf. Therefore, no encoding of a character can be a prefix of an encoding of another character (prefix code).

Huffman Coding Trees

To determine the optimal (shortest) encoding for a given string, we first have to find the frequencies of characters in that string.

Let us consider the following string:

eeadfeejjeggebeeggddehhhececddeciedee

It contains 1×a, 1×b, 3×c, 6×d, 15×e, 1×f, 4×g, 3×h, 1×i, and 2×j.

We can now use **Huffman's** algorithm to build the optimal coding tree.

Huffman Coding Trees

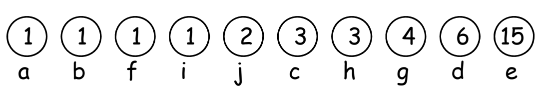
For an alphabet containing n letters, Huffman's algorithm **starts with n vertices**, one for each letter, labeled with that letter and its frequency.

We then determine the **two vertices** with the lowest frequencies and replace them with a **tree** whose root is labeled with the **sum** of these two frequencies and whose two children are the two vertices that we replaced.

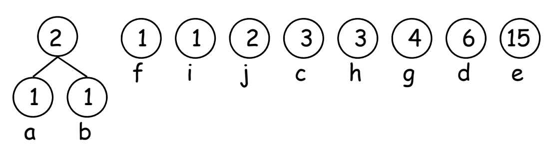
In the following steps, we determine the two lowest frequencies among the **single vertices and the roots of trees** that we already created.

This is repeated until we obtain a **single tree**.

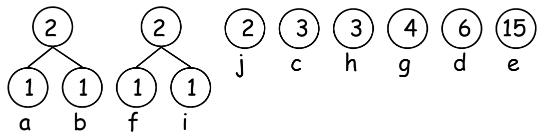
Huffman Coding Trees



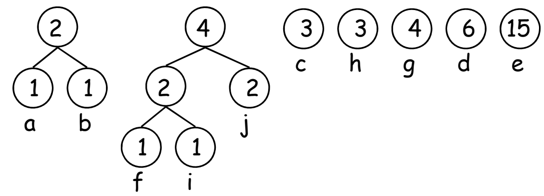
Huffman Coding Trees



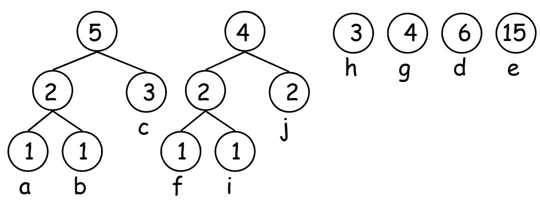
Huffman Coding Trees



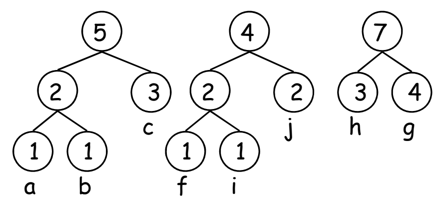
Huffman Coding Trees



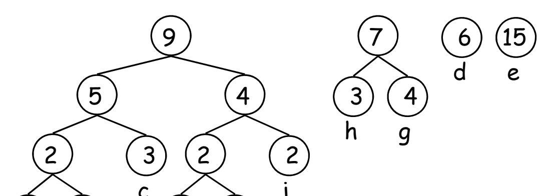
Huffman Coding Trees



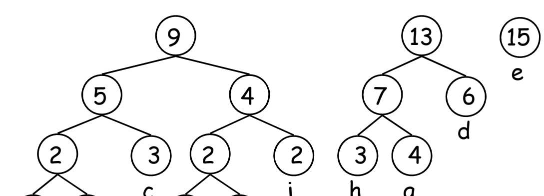
Huffman Coding Trees



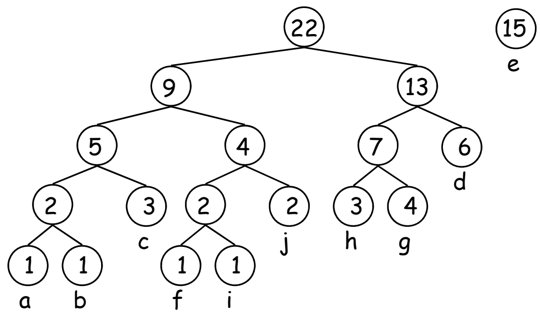
Huffman Coding Trees



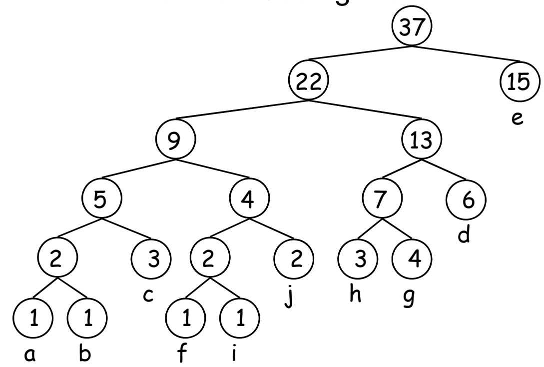
Huffman Coding Trees



Huffman Coding Trees

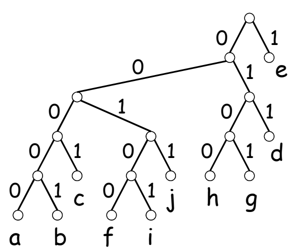


Huffman Coding Trees



Huffman Coding Trees

Finally, we convert the tree into a **prefix code tree**:



The **variable-length codes** are:

a (freq. 1): 00000
b (freq. 1): 00001
c (freq. 3): 0001
d (freq. 6): 011
e (freq. 15): 1
f (freq. 1): 00100
g (freq. 4): 0101
h (freq. 3): 0100
i (freq. 1): 00101
j (freq. 2): 0011

Huffman Coding Trees

If we encode the original string

eeadfeejjeggebeeggddehhhececddeciedee

using a fixed-length code, we need four bits per character (for ten different characters). Therefore, the encoding of the entire string is $4 \cdot 37 = 148$ bits long.

With our variable-length code, we only need $1 \cdot 5 + 1 \cdot 5 + 3 \cdot 4 + 6 \cdot 3 + 15 \cdot 1 + 1 \cdot 5 + 4 \cdot 4 + 3 \cdot 4 + 1 \cdot 5 + 2 \cdot 4 = 101$ bits.

Huffman Coding Trees

It can be shown that, for any given string, Huffman coding trees always produce a variable-length code with **minimum description length** for that string.

For more on Huffman's algorithm, please take a look at:

http://www.cs.duke.edu/csed/poop/huff/info/

Tree Universal Address System

In trees, the order of children from left to right is often important and must be fixed.

In the Universal Address System each vertex has an address like 2.3.4.1

- The root has address 0.
- The n children of the root are labeled 1 to n, left to right.
- The m children of a vertex labeled A are labeled A.1, A.2, …, A.m, left to right.

Thus 2.3.4 would be the fourth child of the third child of the second child of the root (left to right in each case).

Object Identifiers

An example of this is the OID system, object identifiers.

These are used as a universal means of describing objects.

See http://www.alvestrand.no/objectid/

Tree Traversal

There are several schemes for systematically visiting all vertices of a tree. See section 11.3.

Generally when we visit a vertex we do something at the vertex, such as computing something or outputting some value.

Preorder Traversal

In preorder traversal of a tree,

1. We visit the root first.
2. Next we visit the subtrees (if any) T_1, T_2, …, T_n left to right, visiting each subtree in preorder.

Inorder Traversal

In inorder traversal of a tree,

1. We visit the left subtree T_1 first, if it exists, applying inorder traversal to it.
2. We visit the root next.
3. Next we visit the remaining subtrees (if any) T_2, …, T_n left to right, visiting each subtree using inorder.

Postorder Traversal

In postorder traversal of a tree,

1. We visit the the subtrees (if any) T_1, T_2, …, T_n left to right, visiting each subtree in postorder.
2. Last, we visit the root.

Tree Traversals and Arithmetic Expressions

Arithmetic expressions such as (x+y)*(yx -z) are commonly stored in trees for evaluation.

The infix form (x+y)*((y*x) -z) would come from an inorder traversal of the tree.

The prefix or Polish Notation form would be *+xy-*yxz (preorder traversal of the tree).

The postfix or Reverse Polish Notation (RPN) form would be xy+yx*z-* (postorder traversal)

The latter two forms don't need parentheses, though you have to know where the numerical symbols start and end.