Section 3.6

Binary Trees

ROOTED AND LEVELED TREES

- All trees in this section are rooted.
- The trees will also be **leveled**, that is, the root r will constitute level 0, the neighbors of r will constitute level 1, the neighbors of the vertices on level 1 that have not yet been placed in a level will constitute level 2, etc.
- If v is a vertex in level k, the neighbors of v in level $k + 1$ are called the **children** (or **descendants**) of v.
- The neighbor of v on level $k - 1$ (if it exists) is called the **parent** (or **father** or **predecessor**) of v.
- The **height** of a leveled tree is the length of a longest path from the root to a leaf, or alternatively, the largest level number of any vertex.

BINARY TREES

- A **binary tree** is a rooted, leveled tree in which any vertex as at most two children.
- The descendants of v are referred to as the **left child** and **right child** of v.
- Binary trees are said to be **ordered** because of this structure.
- If every vertex of a binary tree has either two children or no children, then we say it is a **full** binary tree.

DISTINCT BINARY TREES

Due to the structure imposed in designating a distinction between the left and right child, we obtain distinct binary trees when ordinary graph isomorphism holds.

TRAVERSAL OF TREES

- In order to recover the information stored in binary tree, we must visit the vertices of the tree in an order that allows us to retrieve the information and understand how that data are related.
- This process is a called a **tree traversal**.
- This is accomplished with the aid of the tree structure and a particular set of rules for deciding which neighbor to visit next.

INORDER TRAVERSAL
(AKA SYMMETRIC ORDER TRAVERSAL)

Algorithm 3.6.1 Inorder Traversal of a Binary Tree.

Input: A binary tree $T = (V, E)$ with root r.

Output: An ordering of the vertices of T (that is, the data contained within these vertices, received in the order of the vertices.)

Method: Here "visit" the vertex simply means perform the operation of your choice on the data contained in the vertices.

PROCEDURE INORDER

```
**Procedure inorder**($r$)
If $T \neq \emptyset$, then,
    inorder(left child of $v$)
    visit the present vertex
    inorder(right child of $v$)
end
```

COMMENT ON INORDER PROCEDURE

The recursive algorithm 3.6.1 performs the following at each vertex.

1. Go to the left child if possible.
2. Visit the vertex
3. Go to the right child if possible.

PREORDER AND POSTORDER TRAVERSAL

- In **preorder traversal**, we visit the vertex, go to the left child, and then go to the right child.
- In **postorder traversal**, we go to the left child, go the right child, and, finally, visit the vertex.

HUFFMAN CODING

A Huffman code is a way to reduce the number of bits needed to store characters while still maintaining the characters in the message. It is very useful when there the number of available characters is few, their frequencies (or probabilities) are known (or at least approximately known), and there is a large string of characters to store. (An example would be from bioinformatics with DNA sequences.) The data is usually encoded in the form of a binary number (that is, a sequence of 0s and 1s.)

IDEA BEHIND HUFFMAN CODE

The main idea behind the Huffman code is to use fewer bits to store characters that occur frequently and use more bits to store the characters that occur less frequently.

Huffman code uses a binary tree to encode and decode the string. We do this by placing a value of 0 on the edge from any vertex to its left child and a value of 1 on any edge from a vertex to its right child. Such a tree is called a **Huffman tree**.

CONSTRUCTION OF A HUFFMAN TREE

Algorithm 3.6.2 Construction of a Huffman Tree.

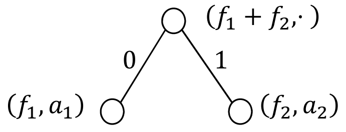
Input: Ordered pairs consisting of the frequencies f_i and the characters a_i corresponding to the frequencies in ascending order of frequencies. That is, $((f_1, a_1), (f_2, a_2), \dots (f_n, a_n))$.

Output: An Huffman tree with leaves corresponding to the ordered pairs above.

Method: From a collections of trees, select the two root vertices corresponding to the smallest frequencies. Then insert a new vertex and make the two selected vertices the children of this new vertex. Return this tree to the collection of trees and repeat this process until only one tree remains.

CONSTRUCTION OF A HUFFMAN TREE (CONTINUED)

1. If $n = 2$, then halt, thereby forming the tree:



CONSTRUCTION OF A HUFFMAN TREE (CONCLUDED)

2. If $n > 2$, then reorder the frequencies so that f_1 and f_2 are the smallest frequencies. Let T_1 be the Huffman tree resulting from the algorithm being recursively applied to the frequencies $(f_1 + f_2, f_3, \dots, f_n)$ and let T_2 be the Huffman tree that results from calling the algorithm recursively on the frequencies (f_1, f_2). Halt the algorithm with the tree that results from substituting T_2 for some leaf of T_1 (which has value $f_1 + f_2$).

WEIGHTED PATH LENGTH

The **weighted path** length for the encoding tree is defined to be

$$\sum_{1 \le i \le n} f_i l_i$$

where f_i is the frequency of the ith letter and l_i is the length of the path from the root in the Huffman tree corresponding to the ith letter.

The average length of each character encoded can be obtained by taking the weighted path length and dividing by $\sum_{i=1}^{n} f_i$ (the number of characters being encoded).

HUFFMAN TREES ARE "OPTIMAL"

Theorem 3.6.1: A Huffman tree for the frequencies $(f_1, f_2, \ldots, f_n)$ has minimum weighted path length among all full binary trees with leaves $f_1, f_2, \ldots, f_n$.

Theorem 3.6.2: If $c_1, c_2, \ldots, c_n$ are the binary codes assigned by Huffman's algorithm to the characters with frequencies $f_1, f_2, \ldots, f_n$, respectively, and if $f_i < f_j$, then $length(c_i) \geq length(c_j)$.