BINARY TREES : Guided Exercises

1. Write recursive and non-recursive algorithms to determine in a binary tree:
(a) The number of nodes.
(b) The number of leaves.
(c) The sum of the contents of all nodes.
(d) The depth.

2. Write an algorithm to determine whether a binary tree is:
(a) Strictly binary.
(b) Complete.

3. Develop algorithms to find duplicates in a sequence of n numbers:
(a) Using a linked list.
(b) Using a binary search tree.
Count the number of comparisons in each case.

4. Find recursive algorithms for traversing a binary tree.

5. Find algorithms for searching, inserting, and deleting in a binary search tree.

6. Write a recursive algorithm and an iterative one to search for the leftest leaf.

7. Representation of a Linked List by a Tree:
We can represent a linked list by a binary tree in the following way: The elements of the linked list are at the level of the leaves. Each non-leaf node contains the count of leaves in its left subtree.
7.1 Find the K-th element of the linked list represented in this way.
7.2 Study and write the construction algorithm.
7.3 Study and write the deletion algorithm.

8. Implementation of Trees:
8.1 Implement the abstract machine defined on trees using the following representations:
(a) Static representation.
(b) Sequential static representation.

8.2 A binary tree can be represented as follows:
If A is an array with n elements, A(i) = j if j is the parent of node i. A(i) = 0 if i is the root. This is called the parent representation. Translate the operations of the abstract machine defined on binary trees.

8.3 Representation of trees by means of linked lists of children:

We have an array T of linked lists indexed by nodes, assumed to be numbered from 1 to n. Each linked list pointed to by T(i) contains the children of node i.
Implement the abstract machine.

10. Huffman's Algorithm

Suppose we have messages composed of sequences of characters. In each message, the characters are independent and appear with known probabilities. We want to encode each character into a sequence of 0's and 1's such that no code is the prefix of another code. This prefix property allows us to decode a string of 0's and 1's. To achieve this, we will implement Huffman's algorithm using the following data structures:

1) An array TREE, where each element is a triplet (Index in TREE of the left child, Index in TREE of the right child, Index in TREE of the parent) representing binary trees. The Parent field allows us to find paths from the leaves to the root, enabling us to determine the code for each character.
2) An array ALPHABET, where each element is a tuple (Symbol, Probability, Leaf). Leaf is the index in the TREE array. This array associates each symbol of the alphabet to be encoded with its corresponding leaf and stores the probability for each character.
3) An array FOREST of elements representing trees. Each element is a tuple (Weight, Root).

10.1 Initialization of the FOREST, ALPHABET, and TREE arrays.

10.2 Pseudo-Algorithm (Outline) with necessary Modules:
Initialize FOREST with individual trees for each character.
Initialize ALPHABET with symbols, probabilities, and leaf indices.
Initialize TREE with empty trees.
While FOREST has more than one element:
  Choose two trees from FOREST with the lowest weights.
  Create a new tree with these two trees as children.
  Update FOREST with the new tree and its weight.
  Update the parent indices in TREE.
Construct Huffman Codes from TREE.

10.3 Module Development: Implement the initialization and code construction modules as needed.
10.4 Huffman Algorithm: Provide the complete Huffman algorithm based on the outline and modules developed.
10.5 Application: Write the algorithm for archiving a program written in PASCAL using the Huffman code, and then provide the algorithm to unarchive it.