```cpp
#include <iostream>
#include "Fib_Tree.h"



//////////////////////////////////////////////////////////////////////
//                        METHODS
//////////////////////////////////////////////////////////////////////


// Constructor - Initially it's "empty"

Tree::Tree(void) {
    root  = nullptr;
    size  = 0;
    depth = 0;
    leafs = 0;
}


//*********************************************************************

// Destructor: wraps another method 'empty' (see below).
// It is probably unnecessary because through the pointers that link
// the nodes each node will call its own destructor (if the structure
// has one). Scanning the Tree recursively is for extra safety.


Tree::~Tree(void) {
    empty(root);
}


//***********************************************************************

// Recursively delete every node, starting from the "leaves".
// A leaf is a node representing the F(0) and F(1) numbers in the
// Fibonacci's sequence.

// NOTE: if the tree is built correctly, every node branches out in 2
// directions except for the leaves, which have no links (L AND R are
// nullptr). Therefore it's safe to check only one direction, L for
// example, to identify the leaves.


void Tree::empty(Node * node) {
    if (node->L) {                  // Not a leaf
        empty(node->L);
        empty(node->R);
        delete node;
    }
    else {                          // A Leaf
        delete node;
        return;
    }
}


```

```
//*************************************************************************

// It calls itself recursively to generate the sequence until F(n).
// When n >= 2, fib(n , ...) causes two calls, generating 2 branches
// towards lower numbers in the sequence (via the pointers L and R) until
// the calls to fib(0 , ...) or fib(1 , ...).
// These calls are the leaves, so they set L and R to nullptr.


void Tree::fib(int n, Node* node) {
    // Leaves
    if (n == 0) {
        node->val = 0;
        node->L   = nullptr;
        node->R   = nullptr;
        leafs++;
    }
    else if (n == 1) {
        node->val = 1;
        node->L   = nullptr;
        node->R   = nullptr;
        leafs++;
    }
    // n >= 2 - Not leaves
    else {
        // Create 2 branches
        Node * newNodeL = new Node;
        Node * newNodeR = new Node;
        size += 2;

        // Connect 2 branches
        node->L = newNodeL;
        node->R = newNodeR;

        // Build children nodes
        fib(n - 1, newNodeL);
        fib(n - 2, newNodeR);
        node->val = newNodeL->val + newNodeR->val;
    }
}



//************************************************************************

// Observation: the depth of the tree is simply equal to n since the
// slowest branch (at the left of the root) decreases by 1 until F(0).
// BUT if n = 0, a depth of 1 must be set explicitly otherwise the code
// will leave it to 0.


void Tree::buildTree(int n) {
    depth = (n == 0) ? 1 : n;
    size = 1;
    root = new Node;              // Start the tree!
    fib(n, root);                 // Build the tree!
}
```

```cpp

//**************************************************************************
// Recursively proceeds from the root down to the leaves.
// Along the way it prints 'val'. This results in the pre-ordered printing.


void Tree::printNode(Node * node) {
    if (node->L) {                                   // Not a leaf
        std::cout << node->val << ' ';
        printNode(node->L);
        printNode(node->R);
    }
    else                                             // A leaf
        std::cout << node->val << ' ';
}



//****************************************************************************

// Prints the tree in pre-order and the information of interest.


void Tree::printTree(void) {
    std::cout << "Call tree in pre-order: ";
    printNode(root);
    std::cout << endl;
    std::cout << "Call tree size: "  << size  << endl;
    std::cout << "Call tree depth: " << depth << endl;
    std::cout << "Call tree leafs: " << leafs << endl;
}
```