Q1: Implement the method printDFS(), which should print the labels of the tree’s nodes in preorder depth-first-search (DFS). The labels should be separated by new-lines and may be printed using toString() method.

For example, for the graph below, the method should print the following lines:

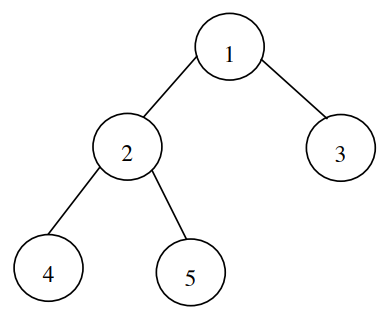
1

2

4

5

3



What is the complexity of this method?

#include <iostream>

using namespace std;

struct Node {

int val;

Node\* left;

Node\* right;

Node(int v) : val(v), left(nullptr), right(nullptr) {}

};

class Tree {

public:

Node\* root;

Tree(int rootValue) {

root = new Node(rootValue);

}

void insertChild(Node\* parent, int val) {

Node\* child = new Node(val);

if (parent->left == nullptr) {

parent->left = child;

}

else if (parent->right == nullptr) {

parent->right = child;

}

}

void printDFS(Node\* troot) {

if (troot != nullptr) {

cout << troot->val << endl;

printDFS(troot->left);

printDFS(troot->right);

}

}

};

int main() {

Tree tree(1);

tree.insertChild(tree.root, 2);

tree.insertChild(tree.root, 3);

tree.insertChild(tree.root->left, 4);

tree.insertChild(tree.root->left, 5);

tree.printDFS(tree.root);

return 0;

}

A screenshot of a computer

Description automatically generated  
The **insertChild** method's efficiency in the provided code depends on how spread out or compact the tree is. In a well-arranged tree, adding a child takes a reasonably short amount of time, approximately log(n) where 'n' is the number of items in the tree. However, if the tree is not well-organized, like a long chain, adding a child could take a longer time, specifically n steps, where 'n' is the total number of items. The memory needed for this method is proportional to the number of items, and in the worst-case scenario, it could be as much as 'n'. To ensure quick and consistent performance, especially when dealing with unsorted data, using balanced tree structures can be beneficial.

Top of Form

Q2: Implement the method insertBST(Item i). The method assumes that the tree is a binary search tree (BST) and inserts item ***i***into it. When the item is inserted, the method should modify the rest of the tree so that the tree is still a BST. If the item already exists in the tree, another copy should be inserted.

#include <iostream>

using namespace std;

class TreeNode {

public:

int value;

TreeNode\* left;

TreeNode\* right;

TreeNode(int val) : value(val), left(nullptr), right(nullptr) {}

};

class BinarySearchTree {

private:

TreeNode\* root;

TreeNode\* insertRecursive(TreeNode\* root, int item) {

if (root == nullptr) {

return new TreeNode(item);

}

if (item < root->value) {

root->left = insertRecursive(root->left, item);

}

else if (item > root->value) {

root->right = insertRecursive(root->right, item);

}

else {

root->right = insertRecursive(root->right, item);

}

return root;

}

void inorderTraversal(TreeNode\* root) {

if (root != nullptr) {

inorderTraversal(root->left);

cout << root->value << " ";

inorderTraversal(root->right);

}

}

public:

BinarySearchTree() : root(nullptr) {}

void insertBST(int item) {

root = insertRecursive(root, item);

}

void traverseInorder() {

cout << "Inorder Traversal: ";

inorderTraversal(root);

cout << endl;

}

};

int main() {

BinarySearchTree bst;

bst.insertBST(50);

bst.insertBST(30);

bst.insertBST(70);

bst.insertBST(20);

bst.insertBST(40);

bst.traverseInorder();

return 0;

}

A screenshot of a computer

Description automatically generated