Workshop Exercises 7

Exercise 1: Dodgy Trees

Recall the `TreeNode` struct for nodes in a binary tree:

```
struct TreeNode{
    int data;
    TreeNode *leftChild, *rightChild;
};
```

In this exercise, we want to implement a function which gets a pointer to a `TreeNode` and needs to check if it is the root of a proper sorted binary tree.

Base case

Implement the function

```
bool isSorted(TreeNode *root){
/* ... */
}
```

The function should recursively check whether the tree is sorted, i.e.

- if `leftChild` is not null, the subtree below `leftChild` is sorted and only contains numbers less than `data`
- if `rightChild` is not null, the subtree below `rightChild` is sorted and only contains numbers greater than `data`

You will probably need some auxiliary functions as well.

Example Solution

We could break this problem down in several different ways: I will give one solution using additional functions `allLess` and `allGreater`. As an added exercise, see if you can do it with fewer (Look at how similar `allLess` and `allGreater` are. Maybe you can combine them? How?)

```
// Check whether all values in the subtree below n are less than the pivot
bool allLess(TreeNode *n, int pivot){
    if(n != nullptr)
        return ((n->data < pivot) &&
                allLess(n->leftChild, pivot) &&
                allLess(n->rightChild, pivot));
    // otherwise: nothing to check
    return true;
}

// Check whether all values in the subtree below n are greater than the pivot
bool allGreater(TreeNode *n, int pivot){
    if(n != nullptr)
        return ((n->data > pivot) &&
                allGreater(n->leftChild, pivot) &&
                allGreater(n->rightChild, pivot));
    // otherwise: nothing to check
    return true;
}

bool isSorted(TreeNode *root){
    if(root != nullptr)
        return (isSorted(root->leftChild) &&
                isSorted(root->rightChild)&&
                allLess(root->leftChild, root->data) &&
                allGreater(root->rightChild, root->data));
    // otherwise: nothing to check
    return true;
}
```

Advanced case

The base solution can be sabotaged by constructing a mal-formed "tree", for example like this:

```
void naughty(){
    TreeNode n1 = {1, nullptr, nullptr};
    TreeNode n0 = {0, nullptr, &n1};
    n1.leftChild = &n0;
    isSorted(&n0);
}
```

This will cause an infinite recursive descent (can you see why), leading to an eventual crash when the program runs out of stack space. Let us try to prevent this. Modify your `isSorted` function using

- a set of node pointers to keep track of which nodes have already been seen
- exceptions to signal an error in the tree structure, in case we re-encounter a node that is already in the set.
- possibly some more auxiliary functions

so that it produces an error message in case of shenanigans like above (and still works normally otherwise).

Example Solution

We can re-use the `allLess` and `allGreater` functions from the base solution. We split the actual `isSorted` function into the recursive part and the one that takes care of setting up the set of known nodes, and handling exceptions:

```
// A custom exception struct
struct MalformedTree{};

bool isSortedRec(TreeNode *root, std::set<TreeNode*> &seen){
    if(root != nullptr)
        if(seen.find(root) != seen.end())  // We have been here before
            throw MalformedTree{};
        else{
            seen.insert(root);             // Remember that we have been here
            return (isSortedRec(root->leftChild, seen) &&
                    isSortedRec(root->rightChild, seen)&&
                    allLess(root->leftChild, root->data) &&
                    allGreater(root->rightChild, root->data));
        }
    // otherwise: nothing to check
    return true;
}

bool isSortedFixed(TreeNode *root){
    std::set<TreeNode*> seen;
    try{
        return isSortedRec(root, seen);
    }
    catch(MalformedTree &e){
        // How to actually handle this case will depend on the bigger context
        std::cout << "Nice try" << std::endl;
    }
}
```