ECE-250 – Algorithms and Data Structures (Winter 2012)

Tutorial 4 (2012-02-09) – Suggested solutions

1 - Given two nodes in a binary tree (the two nodes are assumed to be nodes inside the same tree):

Write a C++ function that determines if one is an ancestor of the other one, and discuss the runtime of such function for the following two cases:

(a) The definition of the tree includes a member parent.
(b) Does not include a member parent.

In both cases, the function declaration would be as follows:

```
template <typename Type>
bool ancestor (const Binary_node<Type> * node1, const Binary_node<Type> * node2);
```

Solution:

The main difference between the two options is that if we have the parent member, we can do a linear search (testing whether the other node is in the path from the root to this node), and thus, the runtime is $\Theta(h)$, where h is the height of the tree. Otherwise, we have to do a recursive traversal of all descendants of a node, to see if the other node is included; runtime being $\Theta(n)$, where n is the number of nodes.

Notice that in both cases, we have to repeat the same procedure in both directions — we start from node1 to see if node2 matches, and if it doesn't, we need to repeat, starting from node2, to see if node1 matches!

The code would be:

(a)

```
template <typename Type>
bool ancestor (const Binary_node<Type> * node1, const Binary_node<Type> * node2)
{
    const Binary_node<Type> * n = node1;
    while (n != node2 && ! n->is_root())
    {
        n = n->parent();
    }

    if (n == node2)
    {
        return true;
    }
    else
    {
```

```
        n = node2;
        while (n != node1 && ! n->is_root())
        {
            n = n->parent();
        }

        return n == node1;
    }
}
```

BTW... Why const Binary_node for the local variable n? (why const for the parameters?)

(b)

```
template <typename Type>
bool ancestor (const Binary_node<Type> * node1, const Binary_node<Type> * node2)
{
    if (node1 == node2)
    {
        return true;
    }

    if (node1 == NULL || node2 == NULL)
    {
        return false;
    }

    return    ancestor (node1->left(), node2)
           || ancestor (node1->right(), node2)
           || ancestor (node2->left(), node1)
           || ancestor (node2->left(), node1);
}
```

2 - Given a general tree, with an implementation similar to the one shown in the course slides, write a function that, given a node, searches the path from the root to that node and returns the smallest value greater than the value in the node.

Solution:

We refer to slide 20 from 2012-02-03.

The problem can be seen as just finding the lowest value in a sequence; keep track of the "lowest so far", and for each value, compare to see if it is lower than the lowest so far, and replace it (the lowest) if it is the case. The one tricky detail is how to initialize that "lowest so far" variable — the typical solution considered the most reasonable option is to initialize it to the first value in the sequence (in a sense, this *is* the lowest we've seen so far, since it is the only one so far).

The problem here is that we don't necessarily have a first value, since we're searching through values that match a condition. And for that matter, there may not be such value, since there may not be any value greater than the value in the node (in which case the *only* reasonable option is to throw an exception). So, we would have to split the loop into two parts: search for the first value that is greater than the value in the node, and then keep searching for the lowest starting from there:

```
template <typename Type>
Type lowest_gt (const Binary_node<Type> * node)
{
    const Binary_node<Type> * n = node;
    while (!n->is_root() && n->retrieve() <= node->retrieve())
    {
        n = n->parent();
    }

    if (n->retrieve() <= node->retrieve())
    {
        throw some_exception();
    }
        // why do we need to check the values?  is it the
        // same if we test if we're at the root, to see that
        // we ended the loop *because* we reached the root?

    Type lowest = n->retrieve();
    while (n != NULL)
    {
        if (n->retrieve() < lowest && n->retrieve() > node->retrieve())
        {
            lowest = n->retrieve();
        }
        n = n->parent();
    }

    return lowest;
}
```

3 - Write a recursive function that determines whether a given binary tree is a binary search tree. The function declaration would be as follows:

```
template <typename Type>
bool is_bst (const Binary_node<Type> * tree);
```

Solution:

We just follow the recursive definition seen in class: a binary tree is a BST if both sub-trees are BSTs and all the values in the left sub-tree are less than the root node and all the values in the right sub-tree are greater than the root node.

To check whether all values are less than, we check whether the largest value is less than the root node, and same idea for greater than (we'll assume that we have those functions — they're really straightforward, as we discussed in class).

```
template <typename Type>
bool is_bst (const Binary_node<Type> * tree)
{
    if (tree == NULL)
    {
        return true;
    }

    return    is_bst(tree->left()) && max_value (tree->left()) < tree->retrieve()
           && is_bst(tree->right()) && min_value (tree->right()) > tree->retrieve();
}
```

4 - A *Weight-balanced binary tree* is defined as follows:

- A binary tree with 1 element is a weight-balanced binary tree.
- A binary tree with n elements is a weight-balanced binary tree if the two sub-trees are weight-balanced binary trees and the number of nodes of each differ by at most 1.

Write a recursive function that determines whether a given binary tree is a weight-balanced binary tree. The prototype of the function would be as follows:

```
template <typename Type>
bool balanced (const Binary_node<Type> * tree);
```

Solution:

Not much else to say — like the previous question, we simply use the recursive definition, which leads quite directly to our recursive function (the weight is essentially the *size*, for which we already have a recursive function in the course slides — 2012-02-06, slide 31):

```
template <typename Type>
bool balanced (const Binary_node<Type> * tree)
{
    if (tree == NULL)
    {
        return true;
```

```
    }

    return    balanced(tree->left()) && balanced(tree->right())
           && std::abs (size(tree->left()) - size(tree->right())) <= 1;
}
```

Do notice that this function is horribly inefficient (in a sense, the previous one was as well, but this one is far worse — can you see why?). We're calculating weights of trees in a horribly redundant way; the weight: at each recursive call to check if BST, we check the weights, which involves recursively computing weights; but then, when we recurse into each of the sub-trees, we compute the weight *again* (and notice that, as a general rule, it won't even be the second time; we'll compute the weight of the same tree many times over!).

We'll get back to this near the end of the course!