Reading Code

• I think there is something to be learned from reading code and observing issues.

• These are excerpts from code that was turned in for Program 2.

• Take Aways:
  – If code gets too complicated, rethink it
  – Think defense. Never follow pointers that could be null. If you remember nothing else today, remember that!!!!
  – Draw yourself a picture to see all the possible cases. Working 99% of the time means it is WRONG.
  – I can teach you how to program well, if you are open to suggestions.
  – Good variable names makes it easier for the maintainer AND easier for you. You’ll be surprised how clear variable names aid your thinking.
  – USE your return values
  – Reread your code. What you think of first may not be the clearest

Max Element
Example 1

```cpp
template <class Etype>
TreeNode<Etype> *
BinarySearchTree<Etype>::maximumEle(TreeNode <Etype> * & t)
{
    if(t->right == NULL)
    {
        return t;
    }
    maximumEle(t->right);
}
```

```cpp
template <class Etype>
TreeNode<Etype> *
BinarySearchTree<Etype>::maximumEle(TreeNode<Etype> * & t)
{
    if(t==NULL || t->right == NULL)
    
    {
    return t;
    }
    maximumEle(t->right);
    
    // Red flag. return value is not being used. “If you ask your little brother for a drink of water, it is RUDE not to drink it.”
}
```
template <class Etype>
TreeNode<Etype> *
BinarySearchTree<Etype>::successor(TreeNode<Etype> * & curr)
{
    // Red flag – never follow a pointer without checking for null
    if (curr->parent == NULL)
        return successor(curr->right);
    // My successor is not my right child’s successor

    if(curr->element == curr->parent->left->element)
        return curr->parent;
    if(curr->element == curr->parent->right->element)
        return successor(curr->parent);
}
template <class Etype>
TreeNode<Etype> * BinarySearchTree<Etype>::successor(TreeNode<Etype> * &t)
{
    Red flag. Never follow a pointer without checking for NULL
    if(t->parent==NULL) return t->right;  // Parent being null has nothing to do with a successor when we have a right child.
    if(t->right != NULL)
    {
        if(t->right->left != NULL)
            return findMin(t->right);
        return t->right;
    }
    if(t->element > t->parent->element)
        return successor(t->parent);
    return t->parent;
}

template <class Etype>
TreeNode<Etype> *
BinarySearchTree<Etype>::successor(TreeNode<Etype> * &t)
{
    if(t->right!=NULL) Check to see t is not NULL
        return minimumEle(t->right);  // Good, nice reuse of code
    if(t->parent->left==t) Check to see t->parent is not null
        return t->parent;
    while(t->parent->right==t) Try to simplify
        t=t->parent;
    return t->parent;
}
template <class Etype>
TreeNode<Etype> * BinarySearchTree<Etype>::successor(TreeNode<Etype> * & t) {
    TreeNode<Etype> * temp;
    bool isFound = false;
    if(t->right) {return minimumEle(t->right);}  // Great!
    else  // No need to nest, you returned above.
    {
        while(!isFound)  // Why do we need isFound, we jump out of the loop when we find it
            // This looks dangerous. What if item is NOT found?
        {
            temp = t;  // Child is better name.
            t = t->parent;
            if(t->left == temp) { // Dangerous. t could be null. Think DEFENSIVE. Never follow a
                // pointer that could be NULL. This simple rule will solve 90% of pointer problems.
                isFound = true;
                return t;
            }
        }
    }
}

template <class Etype>
TreeNode<Etype> * BinarySearchTree<Etype>::successor(TreeNode<Etype> * t) {
    if (t==NULL) return NULL;
    if (t->right != NULL)
        return minimumEle(t->right);
    TreeNode<Etype> * ancestor = t->parent;  // Nice variable names
    while (ancestor != NULL && ancestor->element< t->element)
    {
        ancestor = ancestor->parent;
    }
    return ancestor;
}
Level Count

template <class Etype>
int BinarySearchTree<Etype>::levelCount(TreeNode<Etype> * 
   &t, int level)
{
int counter=0;  // Not my favorite use of counter

if(t==NULL) return NULL;
if(level!=0)
{
   counter+=levelCount(t->left, level-1);
   counter+=levelCount(t->right, level-1);
}
if(level==0) // Would be part of else, right?
{
   counter++;
}

return counter;
}
template <class Etype>
int BinarySearchTree<Etype>::levelCount(TreeNode<Etype> * & t, int level)
{
    if(t==NULL) return 0;
    if(level==0)
    {
        return 1;
    }
    return levelCount(t->right, level-1) + levelCount(t->left, level-1);
}
template <class Etype>
int BinarySearchTree<Etype>::count(TreeNode<Etype> *t)
{
    if(t==NULL)
        return 0;
    if(t->left != NULL || t->right != NULL)
        return 1 + count(t->left) + count(t->right);
        // Works, but you don’t need the special case. You obviously
        // know you can call it with a null pointer, as this does that, right?
    else
        return 1;
}

FindKthInorder
template <class Etype>
TreeNode<Etype> * BinarySearchTree<Etype>::findKthInOrder(TreeNode<Etype> * t, int k)
{
  // It is inefficient to keep calling count. It multiplies the work done at each call.
  if(k > count(t)) return NULL;
  if((count(t->left) + 1) == k)
    return t;
  if(count(t->left) < k)
    return findKthInOrder(t->right, k-(count(t->left)+1));
  if(count(t->left) >= k)
    return findKthInOrder(t->left, k);

  // No need for last "if", right? We must return a value, so if we get here it is bad.
}

template <class Etype>
TreeNode<Etype> * BinarySearchTree<Etype>::findKthInOrder(TreeNode<Etype> * t, int k)
{
  // Very elegant EXCEPT for the multiple calls to count.
  if(k > count(t)) return NULL;
  if((count(t->left) + 1) == k)
    return t;
  if(count(t->left) < k)
    { return findKthInOrder(t->right, k-(count(t->left) + 1)); }
  return findKthInOrder(t->left, k);
}
Elegant! – but be careful in figuring complexity

template <class Etype>
TreeNode<Etype> *
    BinarySearchTree<Etype>::findKthInOrder(TreeNode<Etype> * t, int k)
{ t = minimumEle();

    for (int i = 0; i < k; i++)
        t = successor(t);
    return t;
}

Diameter
template <class Etype>
int BinarySearchTree<Etype>::diameter(TreeNode<Etype> * t, int &height)
{
    if(t==NULL) return 0;
    int x, y, dia, height1;  // BAD variable names makes reading tougher than needed
    height1=height;  // Confusing. Height comes OUT not goes in.
    x=diameter(t->left, height);
    y=diameter(t->right, height1);

    diam=height1+height + 1;
    height = max (height, height1) + 1;  // Better to have two different names for my height and the height of my left child. Variables are cheap. Understandability is worth the price of an additional variable.

    return max(max(x,y),diam);
}

template <class Etype>
int BinarySearchTree<Etype>::DiameterHeightR(TreeNode<Etype> *t, int &height) // height is coming OUT not being sent in. Use it that way.
{
    if (t == NULL) {  // height = 0; return 0; }
    height is coming OUT not being sent in. Use it that way.
    int rightDiameter = DiameterHeightR(t->right, ++heightRight);  // The use of ++ in a argument is confusing. avoid it
    int leftDiameter = DiameterHeightR(t->left, ++heightLeft);  // Nice variable names, clear logic
    int rootDiameter = heightLeft + heightRight + 1;

    //Height is maximum of left/right subtree height
    if (heightLeft >= heightRight)
        height = heightLeft + 1;
    else
        height = heightRight + 1;

    //Return the max between leftDiameter, rightDiameter, and rootDiameter  JUST use a max
    if (leftDiameter >= rightDiameter && leftDiameter >= rootDiameter)
        return leftDiameter;
    else if (rightDiameter >= leftDiameter && rightDiameter >= rootDiameter)
        return rightDiameter;
    else
        return rootDiameter;
}