CS 2420 Program 2 – 20 points
Fall 2014

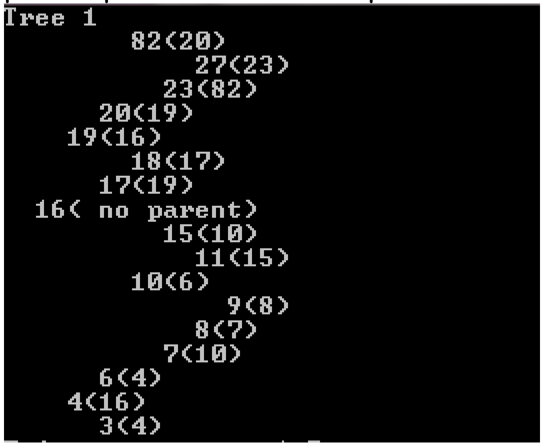
Fun With Recursion

Objective: Fun with recursion. Check the provided starter code to make sure you have the correct prototypes. Feel free to modify prototypes if something else is more useful. You will notice in the starter code I have public helper functions which make it possible to call a routine without knowing the root, but have recursive "worker" routines that depend on knowing the current node. These MUST BE your own work. Do not copy from anywhere.

In the comments to each function, provide a big-Oh expression for the complexity of the functions you write, assuming trees are roughly balanced (depth = log(n) for n nodes). Use recursion where appropriate, but if something isn't logically recursive, don't use recursion.

Documentation of code and identifying the big-Oh is worth four points. Consult the style guidelines.

1. (1 point) Write the function makeEmpty to remove all elements from a tree. Be sure to properly delete the dynamically allocated nodes.
2. (1 points) Write a function, toString, that returns a string containing: a message and the keys (in order) of a binary searchtree, given the root. This should print the tree prettily and show the parent pointers. For example:



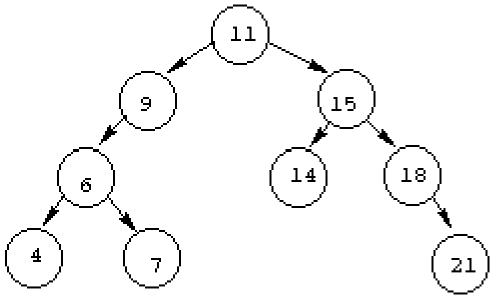
```
Tree 1
        82(20)
            27(23)
          23(82)
      20(19)
    19(16)
        18(17)
      17(19)
  16( no parent)
          15(10)
            11(15)
        10(6)
              9(8)
            8(7)
          7(10)
      6(4)
    4(16)
      3(4)
```

3. (1 points) Write a function, count, that counts the number of nodes in a tree, given the root.
4. (1 points) Write a function, fringe, that returns the count of the leaf nodes of a *binary search tree* rooted at root.
5. (2 points) Write a function to find the inorder predecessor of a node (given a node in the tree). You should use the parent link. This function can be written in fewer than 10 lines of code – but that's a guideline, not a requirement. Note, for each of testing, we find the predecessor of a

value. This needs to use a routine that can find a predecessor from a specific node (without having to traverse the tree from the root).

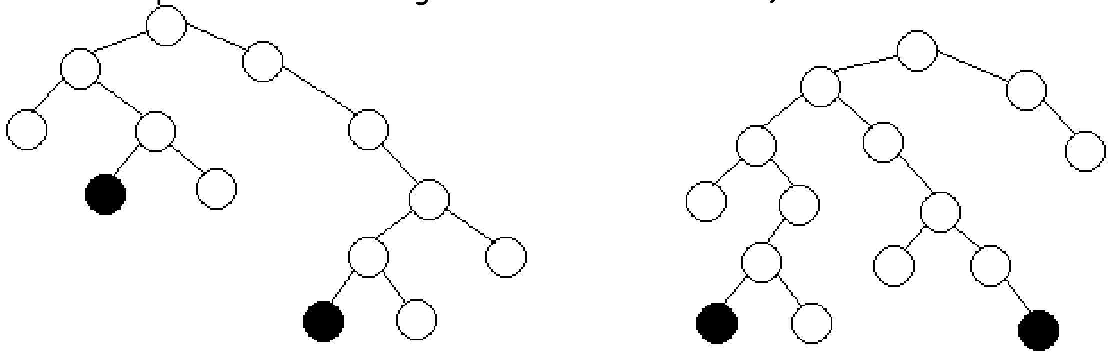
6. (2 points) Write the function *nodesInLevel* that returns the total number of nodes on the specified level. For this problem, the root is at level zero, the root's children are at level one, and for any node, the node's level is one more than its parent's level.

7. (3 points) Write the function *findKthInOrder*. Given an integer k and a binary search tree with unique values, *findKthInOrder* returns a pointer to the node that contains the kth element if the elements are in sorted order – the node with the smallest value is returned if k = 1, the node with the second smallest is returned if k = 2, and so on.
   For example, in the tree shown below (t points to the root), findKthInOrder(t,4) returns a pointer to the node with value 9, findKthInOrder(t,8) returns a pointer to the node with value 18, and findKthInOrder(t,12) returns NULL since there are only 9 nodes in the tree.



   Try to find a RECURSIVE solution, rather than another approach.

8. (3 points) By definition, the *width* of a tree is the number of nodes on the longest path between two leaves in the tree (not considering the direction of the arcs). The diagram below shows two trees each of width nine, the leaves that form the ends of a longest path are shaded (note that there is more than one path in each tree of length nine, but no path longer than nine nodes).
   It can be shown that the width of a tree *T* is the largest of the following quantities:
   - The width of *T*'s left subtree
   - The width of *T*'s right subtree
   - The longest path between leaves that goes through the root of *T* (this can be computed from the heights of the subtrees of *T*)



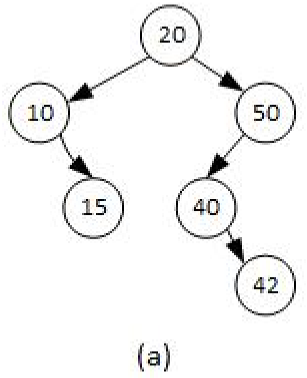
Here's code that's almost a direct translation of the three properties above (assuming the existence of a standard O(1) max function that returns the larger of two values).

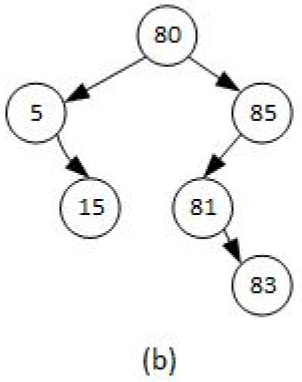
```
 // returns width of tree rooted at  t
int width (TreeNode * t)
 {
    if (t == NULL)  return 0;

    int leftW = width (t->left);
    int rightW = width(t->right);
    int rootW  = height(t->left) + height(t->right) + 1;

    return max(rootW, max(leftW, rightW));
 }
```

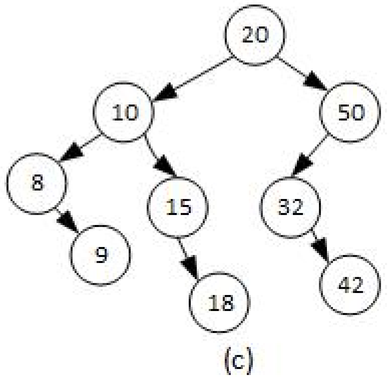
However, the function as written does not run in *O(n)* time because we keep recomputing height. Write a version of width that runs in O(n) tim*e. Hint,* use a function a*s de*scribed below.

```
int widthHeight(TreeNode * t, int & height)

    // pre:  t is a binary tree
    // post: return (via reference param) height = height of t
    //       return as value of function: width of t
    {

    }
```

9. (2 points) Two binary trees *s* and *t* are *isomorphic* if they have the same shape; the values stored in the nodes do not affect whether two trees are isomorphic. In the diagram below, tree a and tree b are isomorphic. Tree c is not isomorphic to the others.

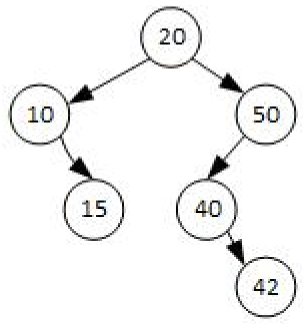




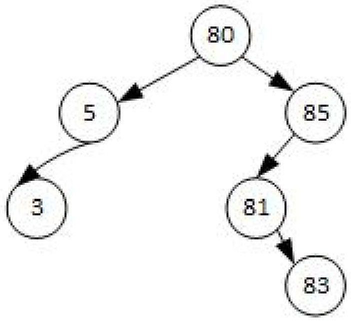


10. (2 points) Two trees *s* and *t* are *quasi-isomorphic* if *s* can be transformed into *t* by swapping left and right children of some of the nodes of *s*. The values in the nodes are not important in determining quasi-isomorphism, only the shape is important. For the trees below tree a and tree b are quasi-isomorphic because if the children of the nodes 10 in tree a on are swapped, the tree having the same shape as tree b is obtained. All three trees below are quasiIsomorphic.

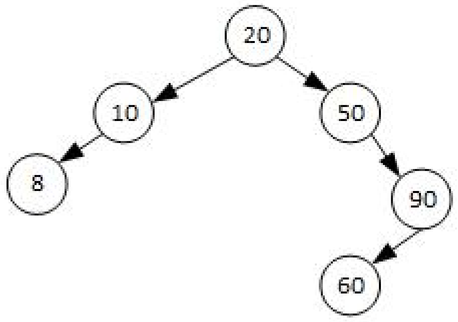
    Write a function *isQuasiIsomorphic* that returns true if two trees are quasi-isomorphic.



(a)

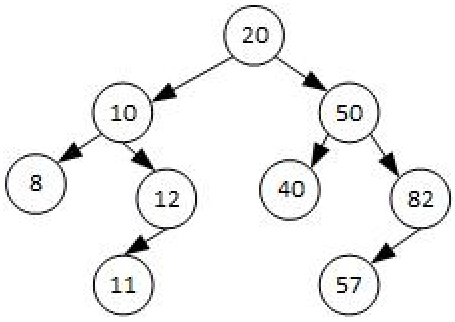


(b)



(c)

11. (2 points) Write a function which returns the least common ancestor of a node in a binary search tree. A least common ancestor is an ancestor of both nodes and is closest to the nodes. A node is considered to be an ancestor of itself. In the tree below, the least common ancestor of 82 and 8 is 20. The least common ancestor of 42 and 50 is 50. The least common ancestor of 57 and 40 is 50.


20
10
50
8
12
40
82
11
57