Data Structures and Algorithms
Homework 9 Solutions (Subject to change)

Written Part

1. (a) contains(x, t)
input: an item x and the root node t to a tree T
result: returns true if T contains x, and false otherwise
```
if T is empty:
    return false;
if (x == t→element):
    return true;
if (x < t→element):
    return contains(x, t→left);
else:
    return contains(x, t→right);
```

(b) rangedPrint(low, high, t)
input: range items low and high as well as the root node t to a tree T
result: print all items in T in range [low, high] in order
```
if T is empty:
    return;
if ( low < t→element ):
    recurse on right subtree;
    return;
if ( high > t→element ):
    recurse on left subtree;
    return;
recurse on left subtree;
print t→element;
recurse on right subtree;
return;
```

(c) stringy(t)
input: the non-null root node t to a BST T
result: BST becomes stringy tree, returns root and leaf of the stringy tree
```
if leaf:
    return {t, t}
if (no left subtree):
    recurse on right
    return { right leaf, t }
if (no right subtree):
    recurse on left and store
    t→left = nullptr;
    left tail→right = t;
    return {left subtree root, t}
recurse on left and store
recurse on right and store
left sub tree tail→right = t;
t→left = nullptr;
t→right = right subtree head
return { left subtree root, right subtree leaf }
```

(d) average_node_depth(t, d)
input: the node t to a BST T and the depth of t
result: returns sum of all node depth and size of the tree in a pair

if (leaf):
    return { d, 1 }
if (left does not exist):
    return { d+right subtree depth, right subtree size + 1 }
if (right does not exist):
    return { d+left subtree depth, left subtree size + 1 }
recurse on left subtree and store
recurse on right subtree and store
return { left subtree sum of depth + right subtree sum of depth + d, size of left subtree + size of right sub tree + 1 }

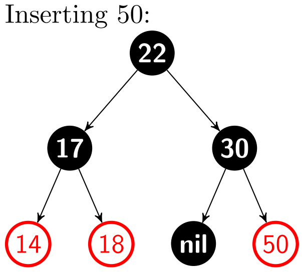
2. The runtime is $O(n)$ where n is the number of nodes in the tree. This is because every iteration pops at least one element from the queue and the queue can have at most n elements during its lifetime.

3. The code is valid according to the implementation done in class. As we know $t \to left$ is valid, t is valid too. Adding an extra check does not break the code.

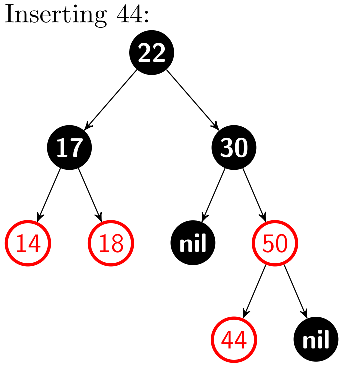
4. The code will compile and run without complain. However, it is incorrect. We need to pass in t by reference. Right now, a new Node is being assigned to a copy of the pointer, but not the pointer itself. Also, after this correction, each node will store incorrect size (one more than actual). This is because when a single node is created, it size is first set to 1 and then incremented again after the conditional checks. The code can be fixed by initializing the size to 0 when created.

```
void insert( const Comparable & x, Node * & t )
{
        if( t == nullptr )
                t = new Node{ x, nullptr , nullptr , 0 };
        else if( x < t->element )
                insert( x, t->left );
        else if( t->element < x )
                insert( x, t->right );
        else
                ; // Duplicate; do nothing
        t->size++;
}
```

5. Inserting 50:

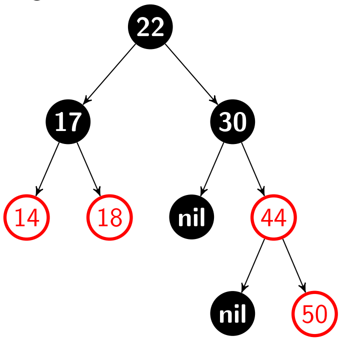


No Issues.
Inserting 44:



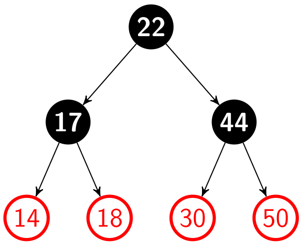
Case 2

(a) Right Rotate



Case 3

(b) Left rotate and recolor:



6.

7.
```
template <class Comparable>
void RedBlackTree<Comparable>::rightRotateRecolor( Node * & k2 )
{
        Node * k1 = k2->left;
        k2->left = k1->right;
        k1->right = k2;
        k2 = k1;
        k2->color = BLACK;
        k2->right->color = RED
}
```

8. The maximum number of pointer changes requires is 6. The most complicated case is a left rotate followed by a right rotate (case 2 turned into case 3), each of which require a change of three pointer.